Achtung!
Weiterhin gilt, der "richtige Loop" an der "richtigen Stelle".

Aufgabe V4.1 Vorbereitungsaufgabe: Vererbung, dynamische Bindung oder auch nicht ;-)

An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `v4x1.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 v4x1 lösen.

Diese Aufgabe ist eine Vorbereitungsaufgabe. Auch wenn die Bearbeitung dieser Aufgabe Pflicht ist, wird Ihre Lösung (vermutlich) nicht im Labor kontrolliert. Sie können aber auf jeden Fall Verständnisfragen im Labor stellen.

Analog zu den in der Vorlesung/Tutorium durchgeführten Übungen finden Sie bei dieser Aufgabe mehrere Klassen vor. Schauen Sie sich alles gründlich an. Befreit von jedweder "Fachlichkeit", trainiert diese Aufgabe die Technik (also die Möglichkeiten von Java ohne irgendeine sinnvolle Anwendung im Hintergrund). Überlegen Sie sich, bevor Sie die Klassen TestFrameAndStarter1 bzw. TestFrameAndStarter2 anstarten, was für eine Ausgabe Sie erwarten und kontrollieren Sie was Sie wirklich bekommen. Klären Sie mögliche Unsicherheiten.

Versuchen Sie auch die Bedeutung von
- ….class
- ….getClass()
- ….getSimpleName()

zu ergründen.

Falls Sie hierzu weitere Informationen in der Java-API (API ::= Application Programming Interface) nachlesen wollen, die Methode getClass() "kommt" aus der Klasse Object und die Methode getSimpleName() "kommt" aus der Klasse Class.
Die API für Java7 finden Sie unter:

`https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html` verständlicher
`https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html` verständlicher

und die API für Java21 finden Sie unter:

`https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Class.html` aktueller
`https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html` aktueller

Das Klassen-Literal .class wird u.a. in der Java Language Specification (15.8.2. Class Literals) beschrieben.
`https://docs.oracle.com/javase/specs/`
bzw.
`https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.8.2`

Aufgabe V4.2 Vorbereitungsaufgabe: Schauen Sie sich "die Thingies/Items" an

An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `v4x2.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 v4x2 lösen.

Diese Aufgabe ist eine Vorbereitungsaufgabe. Auch wenn die Bearbeitung dieser Aufgabe Pflicht ist, wird Ihre Lösung (vermutlich) nicht im Labor kontrolliert. Sie können aber auf jeden Fall Verständnisfragen im Labor stellen.

U.u. ist das `thingy`-Package bereits in der Vorlesung besprochen. Über das `thingy`-Package haben Sie Zugriff auf "Dinge" (die in der Klasse Item modelliert sind). Diese Dinge/Items haben Eigenschaften. Konkret: Farbe, Gewicht, Größe und Wert. Im Zusammenhang mit den Collections (=Zusammenstellung en="Sammlung en"), die u.U. erst noch in der Vorlesung vorgestellt werden, werden die Items in Collections organisiert/verwaltet/abgelegt werden.

Im `thingyDemo`-Package finden Sie die Klassen `StarterForItemDemo1 und StarterForItemDemo2`. Schauen Sie sich zunächst den Code dieser Klassen an und starten Sie dann die Klassen und deren Ausgabe an. Haben Sie dabei die Sicht eines Nutzers bzw. Client des thingy-Package.

Aufgabe A4.1 (Karten-)Hand

Vorbemerkung:
Zum Zeitpunkt der Aufgabenstellung kennen wir noch keine "Collections" und insbesondere noch keine "ArrayList". Für diese Aufgabe dürfen Sie nur Arrays verwenden und insbesondere keine "ArrayList".

Die (Spiel-)Karten, die ein Spieler auf der Hand hält werden auch oft (Karten-)Hand genannt. Schreiben Sie eine Klasse **`Hand`**, die dies unterstützt. Die Klasse soll einen Konstruktor aufweisen, der beliebig viele Karten entgegen nehmen kann. Diese Karten sind dann (zunächst) die Hand.
Weiterhin sollen mit einer Prozedur (ein Mutator) **`add()`** beliebig viele Karten hinzugefügt werden können. Entweder als Karten selbst oder in Form einer anderen Hand.

Die sondierende Funktion **`isSuited()`** soll einen Wahrheitswert abliefern für die Aussage, ob alle Karten von einer Farbe sind. Für die nicht so Mathematik-Sicheren: **`true`** steht für das Aussage, dass es **keine** Karten unterschiedlicher Farbe in der Hand gibt.

Schließlich soll die sondierende Funktion **`getHandCards()`** alle Karten, die in der Hand enthalten sind, liefern (Achtung! Die Karten verbleiben (auch) in der Hand) und
die Prozedur **`setHandCards()`** die aktuelle Hand auf die als Parameter übergebenden beliebig vielen Karten setzen.

Überlegen Sie sich auch, wie Sie mit einer "leeren Hand" bzw. "keinen Karten" umgehen. Dies sind Sonderfälle, die in der Realität auftreten können und müssen unterstützt werden.

Es ist immer schön (und später immer gefordert), wenn Sie sich auch überlegen, wie Sie mit dem Nullpointer **`null`** als/im Parameter umgehen. Sofern Sie "unglückliche Wege gehen" könnte diese Aufgabe bei Verwendung von Varargs nicht-trivial werden. Daher ist eine Umsetzung/Implementierung dieses Aspekts nicht Pflicht.

Falls Sie Vorkenntnisse bzgl. "Sicherheit" haben, es wäre schön im "richtigen" Moment Kopien zu erstellen - dies ist aber ausdrücklich **nicht** gefordert.

Die Implementierung der Methoden **`equals()`**, **`toString()`** und **`hashCode()`** ist für die Klasse **`Hand`** bzw. diese Aufgabe nicht gefordert.

Aufgabe A4.2 Konten

In dieser Aufgabe geht es um Bankkonten (**`BankAccount`**). Es sollen Sparkonten (mit Zinsausschüttung - **`SavingsAccount`**), Girokonten (**`CurrentAccount`**) und Überweisungen zwischen Girokonten unterstützt werden. Für diese Aufgabe gilt! :

- Es wird mit Cent-Beträgen und auf den Cent genau gerechnet.
- Die Bank rundet immer zu ihrem Vorteil (also zum Vorteil der Bank;-).
- Konten dürfen zu **keinem** Zeitpunkt negative Kontostände annehmen.
- Es werden nicht alle "Details" eingefordert. Sie müssen einige "Details" selbst erkennen können. Es sollte als Konsequenz des Erlernten beim Lösen dieser Aufgabe klar sein, was gemeint ist und wie es zu lösen ist.

Implementieren Sie einen Daten-Typ **`BankAccount`**.
Dieser Typ soll einen Konstruktor aufweisen, der eine ID (z.B. IBAN) vom Typ **`String`** und ein Startguthaben in Cent vom Typ **`long`** entgegen nimmt.
Ferner soll es einen zweiten Konstruktor geben, der eine ID (z.B. IBAN) vom Typ **`String`** entgegen nimmt. In diesem Fall soll der Startbetrag 0 sein.
Es soll eine Prozedur **`withdraw()`** für das Abheben eines Betrags in Cent vom Typ **`long`** sowie eine Prozedur **`deposit()`** für das Einzahlen eines Betrags in Cent vom Typ **`long`** geben.

Implementieren Sie einen Daten-Typ **SavingsAccount**, der ein Sparkonto beschreibt. Sparkonten sind Bankkonten! Dieser Typ soll einen Konstruktor aufweisen, der eine ID (z.B. IBAN) vom Typ **String**, ein Startguthaben in Cent vom Typ **long** und einen Zinsatz in Promille vom Typ **int** entgegen nimmt.
Es soll eine Prozedur **giveInterest()** für eine Zinsausschüttung geben. Der Aufruf von **giveInterest()** soll den jeweiligen Kontostand um die Zinsen erhöhen.
Die Funktion **getInterestRate()** soll den Zinsatz abliefern.

Implementieren Sie einen Daten-Typ **CurrentAccount**, der ein Girokonto beschreibt. Girokonten sind Bankkonten! Dieser Typ soll

- einen Konstruktor aufweisen, der eine ID (z.B. IBAN) vom Typ **String** und einen Betrag für eine Standardgebühr in Cent vom Typ **int** entgegen nimmt. (Das Startguthaben auf dem Konto ist in diesem Fall 0).
- einen Konstruktor aufweisen, der eine ID (z.B. IBAN) vom Typ **String**, ein Startguthaben in Cent vom Typ **long** und einen Betrag für eine Standardgebühr in Cent vom Typ **int** entgegen nimmt

Auf/mit Girokonten sind Überweisungen möglich.
Für jedes Abheben/Abbuchen (`withdraw()`) - auch als Teil einer Überweisung - wird die Standardgebühr für das jeweilige Konto fällig, die sofort vom Konto abgezogen wird.
Das Einzahlen/Empfangen einer Überweisung (`deposit()`) wird nicht mit der Standardgebühr belegt.

Implementieren Sie einen Daten-Typ **TransferManager**, dessen Objekte es ermöglichen Geld von einem Girokonto auf ein anderes Girokonto zu überweisen.
Es soll eine Prozedur **transfer()** für das konkrete Überweisen geben. Diese Prozedur soll als Parameter das Quell-Giro-Konto, das Ziel-Giro-Konto und den zu überweisenden Betrag in Cent vom Typ **long** aufweisen.

Berücksichtigen Sie das Erlernte[1] bzgl. private, package-scope, protected und public sowie Getter und Setter und wenden Sie dieses Wissen bei der Implementierung an. In den gestellten Tests stellt sich das Problem, das "etwas" erwartet wird, was nicht explizit in der Aufgabe eingefordert wird (konkret "getAccountBallance()"). Sie dürfen diesen Namen gern per Refactoring im jeweiligen Test ändern, wenn ein anderer Name als Konsequenz Ihrer Implementierung mehr Sinn macht. Der Name sollte sich aber nicht eindeutig verschlechtern. Dieser Absatz ist bewusst "mysteriös" gehalten, da Sie eigentlich alles selber erkennen sollten.

Alle von Ihnen zu implementierenden Klassen, die Bankkonten beschreiben, sollen u.a. die Funktion **toString()** aufweisen.

In der **toString()**-Methode werden Sie vermutlich **String.format()** verwenden wollen. Wie in der Vorlesung besprochen, können Sie Ihr Wissen von **printf()** übertragen bzw. Sie können sich

```
System.out.printf( parameter );
```

als

```
System.out.print(String.format( parameter ));
```

Vorstellen.

Zeichnen Sie ein UML-**Klassen-Diagramm** und halten Sie es bei der "Abnahme" bereit um es auf Aufforderung vorzeigen zu können.

Bemerkung:

Alle Parameter sollen in der Reihenfolge in der jeweiligen Parameterliste implementiert werden in der sie auch oben aufgezählt worden.

Die **toString()**-Methode kann/soll auch genutzt werden um die "internen Werte" zu kontrollieren bzw. diese in dem Ausgabe-/Ergebnis-String aufzuführen.

Die Implementierung der Methoden **equals()** und **hashCode()** ist für die Klasse **Hand** bzw. diese Aufgabe nicht gefordert.

[1] Siehe Vorlesung #15 vw08 cw23 250606

Aufgabe A4.3 Längste (Satzfragment-)Palindrom finden

Vorbemerkung:
Sie dürfen die Ergebnisse der "Vorgänger-Aufgabe" vom vorherigen Aufgabenzettel wieder verwenden bzw. geeignet modifiziert nutzen. (Jedoch müssen Sie spätestens jetzt mit dem Typ `String` arbeiten bzw. `char[]` darf nicht mehr genutzt werden - am Ende der Aufgabe finden Sie Hilfestellungen bzgl. der Klasse String)

Schreiben Sie eine Klasse `PalindromeFinder`, die sowohl einen parameterlosen Konstruktor als auch einen Konstruktor mit einem String als Parameter unterstützen soll.
Die von Ihnen zu schreibende Methode `String getLongestPalindrom()` hat die Aufgabe in einem Text, der entweder

mit dem String-Parameter im Konstruktor

oder

mit der von Ihnen zu schreibenden Methode `void setText( String )`

gesetzt werden kann, das längste "Textfragment-Palindrom" zu bestimmen.
Bei der Bestimmung des "Textfragment-Palindroms" sollen alle(!) Zeichen innerhalb des gegebenen Textes berücksichtigt werden. Lediglich die Groß-/Klein-Schreibung der Buchstaben soll ignoriert werden. Sollte die Lösung nicht eindeutig sein, so können Sie ein beliebiges der existierenden längsten Textfragment-Palindrome als Ergebnis zurückgeben.
Die im "TestFrameAndStarter" gegeben Methode doTest() markiert im ersten Testtext alle "Textfragment-Palindrom" mit einer Mindestlänge von zwei Zeichen. (Für die Markierung wurde abweichend von der Aufgabenstellung eine Mindestlänge von zwei Zeichen gewählt, weil sonst alles hätte markiert werden müssen und damit wäre die Markierung als Verdeutlichung wertlos gewesen)
Schließlich soll die Methode `String getText()` den aktuell zu untersuchenden Text abliefern. Also den Text der entweder über den Konstruktor oder mit setText() gesetzt wurde.

Wir können noch keine Rekursion – rekursive Lösungen werden nicht akzeptiert.

Die Implementierung der Methoden `equals()`, `toString()` und `hashCode()` ist für die Klasse `PalindromeFinder` bzw. diese Aufgabe nicht gefordert.

- Mit **length()** lässt sich die Länge des Strings bestimmen.
- Mit **charAt()** lässt sich das Zeichen an einer bestimmten Position im String bestimmen. Analog zu einem Array hat das erste Zeichen die Position 0 und das letzte Zeichen die Position length()-1.
- Mit **Character.toLowerCase(** … **)** lässt sich Buchstabe in einen Kleinbuchstaben umwandeln.
- Mit string**.toLowerCase()** werd in einem (Ergebnis-)String alle Buchstaben in Kleinbuchstaben "umgewandelt".
- Mit string**.substring(** int beginIndex, int endIndex **)** wird ein Teil-String ausgeschnitten beginnend an der Position "beginIndex" und endend vor der Position "endIndex".

Beispiel-Code:

```
String text = "lalilU";

char zeichen = text.charAt(1);                                  // "text.charAt(1)"          liefert 'a'
int textLength = text.length();                                 // "text.length()"           liefert 6

char kleinesZeichen = Character.toLowerCase( text.charAt(5) ); // "Character.toLowerCase( … )"  wandelt Buchstaben in Kleinbuchstaben bzw. liefert 'u'.
String kleinerText = text.toLowerCase();                        // string.toLowerCase()      liefert kleinen Text bzw. "lalilu".
String teilText = text.subString(2,4);                          // string.subString(…)       liefert Teil-Text bzw. "li".
```

Für Java7 finden Sie die API bzgl. der Klasse String unter:
`https://docs.oracle.com/javase/7/docs/api/java/lang/String.html` verständlicher
und für Java21 unter:
`https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html` aktueller

Weiterhin kennen Sie seit AZ#2 V2.3 die/den charArrayVersusStringDemonstrator.

Freiwillige Zusatzaufgaben

Es folgen freiwillige Zusatzaufgaben. D.h. jede dieser Aufgaben ist freiwillig ;-).
Wenn Sie diese freiwillige Zusatzaufgaben freiwillig lösen, dann haben Sie den "Gewinn", dass Sie mehr geübt haben und dass Sie Ihre Lösung für diese freiwillige Zusatzaufgabe im Labor besprechen können (sofern Zeit ist – Pflichtaufgaben haben Vorrang).

Konkret sind die beiden folgenden Zusatzaufgaben "Steigerungen" gegenüber den Pflichtaufgaben.
Insbesondere Z4.2 ist "DIE AUFGABE für Vererbung". Vermutlich wird Z4.2 in P1 oder P2 noch Pflichtaufgabe werden.

Freiwillige Zusatzaufgabe Z4.1 Körper

Es sollen 3-dimensionale mathematische Körper modelliert werden. Gemessen wird in "Units" (also nicht näher spezifizierten Einheiten). Verwenden Sie `double` als Datentyp für eine einzelne Koordinate. Ein Punkt im 3-dimensionalen Raum hat 3 Koordinaten. Wir setzen ein "normales/ kartesisches Koordinatensystem voraus und wir wollen auf keine besonderen physikalischen Phänomäne wie z.B. schwarze Löscher usw. unterstützen.

Entwickeln Sie u.a. geeignete Referenz-Typen (also Klasse oder Interface) für Punkte (`Point`) und für "mathematische" Körper (`Shape`), Quader (`Cuboid`), Würfel (`Cube`)[2] sowie Kugeln (`Sphere`) im 3-dimensionalen Raum mit den zugehörigen Konstruktoren. Die Referenztypen sind im Package **shape** abzulegen.

- Für die Quader (Achtung Würfel sind auch Quader) speichern Sie die Koordinaten der 8 Ecken.
  Min. ein Konstruktor nimmt ein Array von 8 Eck-Punkten entgegen.
- Für die Kugeln den Mittelpunkt und den Radius.
  Min. ein Konstruktor nimmt als 1.Parameter den Mittelpunkt und als 2.Parameter den Radius entgegen.

Es sollen nur gültige Objekte erzeugt werden.
Wichtig ist, dass bei der Ausgabe von Quader oder Würfel mit `toString()`[3] alle acht Eck-Punkte und bei der Kugel Mittelpunkt und Radius ausgegeben werden.

Entwickeln Sie jeweils geeignete Methoden, die

- die Oberfläche (`double getSurface()`),
- das Volumen (`double getVolume()`) und
- den geometrischen Schwerpunkt/Mittelpunkt (`Point getCenter()`)

des jeweiligen Körpers berechnen.
Jede Klasse soll eine eigene "sinnvolle" Methode `@Override public String toString()` aufweisen (s.o.).

Allgemein gilt, dass die Körper **beliebig** im Raum positioniert sein können. Sie können nur davon ausgehen, dass es sich bei dem dreidimensionalen Raum um einen "normalen" Raum handelt (Euklidischer Raum mit kartesisches Koordinatensystem).

Es ist zu prüfen, ob es sich auch wirklich um einen gültigen Körper der jeweiligen Art handelt. Es ist schwer angeraten diese Prüfung für Quader oder Würfel über die "Streckenlängenverteilung" durchzuführen. Zwischen je 2 der 8 Eckpunkte lässt sich die Streckenlänge berechnen. (Was gilt für derartige Streckenlängen – also Kantenlänge, Oberflächendiagonale oder Raumdiagonale in einem Quader und was in einem Würfel?)
Das folgende Vorgehen ist schwer angeraten:

- Bestimmen Sie die (absoluten) Längen aller (ungerichteten) Strecken, die zwischen den 8 (Eck-)Punkten existieren. (Dies sind 28 ungerichtete Strecken.)
- Sortieren Sie diese Längen. (Hierfür dürfen Sie `Arrays.sort()` verwenden).
- Bestimmen Sie nun mit Hilfe dieser sortieren Längen bzw. der vorliegenden Längenverteilung, ob es sich um einen `Cuboid` (oder einen `Cube`) handelt.

Als Konsequenz von Rundungsfehlern, die bei Rechnungen mit Gleitkommazahlen auftreten können, müssen Sie ein geeignetes `epsilon` bestimmen, das eine obere Grenze für tolerierbare Abweichungen (max. Rundungsfehler) vorgibt. Dieses Epsilon sollte als public Konstante in Shape definiert sein. Ein sinnvoller Wert ist 10^{-12} (zumindest im Rahmen dieser Aufgabenstellung).
Ferner ist eine an equals() angelehnte Methode isAcceptedAsEqual() sinnvoll, die letztlich auf dieses `epsilon` zurückgreift und die nicht "den Contract" erfüllen muss bzw. nicht in "Harmonie" mit hashcode() sein muss.
Konkret wird vom gestellten JUnit-Test für die Klasse Point die Methode

```
public boolean isAcceptedAsEqual( final Object otherObject, final double tolerance )
```

benötigt und als aktueller Parameter für tolerance die zuvor eingeforderte Konstante `Sphere.epsilon` verwendet.
Für die Klassen `Cube`, `Cuboid`, `Sphere` ist die Methode isAcceptedAsEqual() "nur" sinnvoll.

Den JUnit-Test finden Sie im package `test`.

---

[2] Wegen möglicher Weiterentwicklungen soll Cube eine eigene Klasse sein.
[3] Jede Klasse sollte eine `toString()`-Methode aufweisen, die alle Informationen über ein Objekt der Klasse abliefert. Aufbau-Empfehlung und Signatur: `public String toString(){ return String.format( ... ); }`

Freiwillige Zusatzaufgabe Z4.2 Resistance Net

Im Rahmen dieser Aufgabe sollen Sie u.a. geeignete Referenztypen für **ComposedResistor**, **OrdinaryResistor**, **ParallelResistor**, **Potentiometer**, **ResistanceNet**, **Resistor** und **SeriesResistor** implementieren.

- Erstellen Sie ein UML-Klassendiagramm **bevor** Sie zu implementieren anfangen.
  Dieses Klassendiagramm ist bei der Abnahme vorzulegen.

Widerstände (engl. "resistor") sind elektrische Bauteile mit einem Widerstandswert, gemessen in der Einheit Ohm mit dem Einheitenzeichen Ω. Aus Widerständen lassen sich Widerstandsnetze zusammensetzen. Hierfür gelten die folgenden Konstruktionsregeln:

- Ein einzelner Widerstand mit dem Widerstandswert R ist ein Widerstandsnetz, wenn auch ein sehr einfaches.
- Zwei oder mehr Netze mit den Widerstandswerten R_1 bis R_n können in Reihe, das heißt hintereinander, geschaltet werden (siehe hierzu beispielsweise http://de.wikipedia.org/wiki/Reihenschaltung). Die Kombination ist ein neues Netz mit dem Gesamt-Widerstandswert R, der bestimmt ist durch:
  $$R = R_1 + \ldots + R_n$$
- Zwei oder mehr Netze mit den Widerstandswerten R_1 bis R_n können parallel, das heißt nebeneinander, geschaltet werden (siehe hierzu beispielsweise http://de.wikipedia.org/wiki/Parallelschaltung). Die Kombination ist ein neues Netz mit einem Gesamt-Widerstandswert R, der bestimmt ist durch:
  $$R = 1 / (1/R_1 + \ldots + 1/R_n) \quad \text{bzw.:} \quad 1/R = 1/R_1 + \ldots + 1/R_n$$

Implementieren Sie geeignete Referenztypen zur Repräsentation derartiger Schaltungen:

- Definieren Sie einen geeigneten Referenztypen für Widerstandsnetze mit Namen: **ResistanceNet**. Objekte, die diesem Typ genügen müssen zumindest auch die folgenden Methoden aufweisen:
  - **`double getResistance()`**
    Liefert den Gesamtwiderstand des Netzes.
  - **`int getNumberOfResistors()`**
    Liefert die Anzahl an einfachen Widerständen im Netz (in nachfolgender Abbildung1 z.B. 6).
  - **`String getCircuit()`**
    Liefert eine Beschreibung der Schaltung als String. Siehe nachfolgende Abbildung1 für Beispiel.

- Definieren Sie einen geeigneten Referenztypen für Widerstände mit Namen: **Resistor**. Achtung! Jeder Widerstand ist auch ein Widerstandsnetz. **Resistor** muss einen Konstruktor mit der Signatur **`Resistor( String, double )`** aufweisen. Hierbei ist der 1.Parameter der Name des Widerstandes und der 2.Parameter der Widerstandswert gemessen in Ohm. Die zugehörige **`getCircuit()`**-Methode soll nur den Namen des Widerstands liefern.

- Definieren Sie einen geeigneten Referenztypen für zusammengesetzte Widerstandsnetze mit Namen: **ComposedResistor**. Zusammengesetzte Widerstandsnetze sind Widerstandsnetze. Achtung! Weder sind zusammengesetzte_Widerstandsnetze Widerstände noch sind Widerstände zusammengesetzte_Widerstandsnetze. Objekte, die diesem Typ genügen müssen zumindest auch die folgenden Methoden aufweisen:
  - **`ResistanceNet[] getSubNets()`**
    Liefert ein Array der Widerstandsnetze aus denen das zusammengesetzte Widerstandsnetz unmittelbar zusammengesetzt ist in der "Original-Reihenfolge" - also genau in der gleichen Reihenfolge in der diese an den Konstruktor übergeben wurden.
- Definieren Sie einen geeigneten Referenztypen für Reihen-Schaltungen von Widerstandsnetzen (bzw. serielle Widerstandsnetze) mit Namen: **SeriesResistor**.

  Die **`getCircuit()`**-Methode eines seriellen Widerstandsnetzes soll alle unmittelbar enthaltenen Widerstandsnetze als String durch "+" getrennt liefern. Der String / das gesamte Widerstandsnetz ist von Runden Klammern umrahmt:

  Beispiel: ($Widerstandsnetz_1$ + $Widerstandsnetz_2$ + $Widerstandsnetz_3$)

  Achtung! Um mögliche Rundungsfehler bei der Berechnung des Gesamt-Widerstands $R = R_1 + \ldots + R_n$ zu minimieren, müssen Sie in geeigneter Reihenfolge die Werte verknüpfen bzw. an der "entscheidenden Stelle" beginnend mit dem jeweils kleinesten zum größten Wert hin in aufsteigender Reihenfolge aufaddieren. Die ursprüngliche im Konstruktor übergebene Reihenfolge darf jedoch nicht verloren gehen.
- Definieren Sie einen geeigneten Referenztypen für Parallel-Schaltungen von Widerstandsnetzen (bzw. parallele Widerstandsnetze) mit Namen: **ParallelResistor**.

Die **`getCircuit()`**-Methode eines parallele Widerstandsnetzes soll alle unmittelbar enthaltenen Widerstandsnetze als String durch "|" getrennt liefern. Der String / das gesamte Widerstandsnetz ist von Runden Klammern umrahmt:

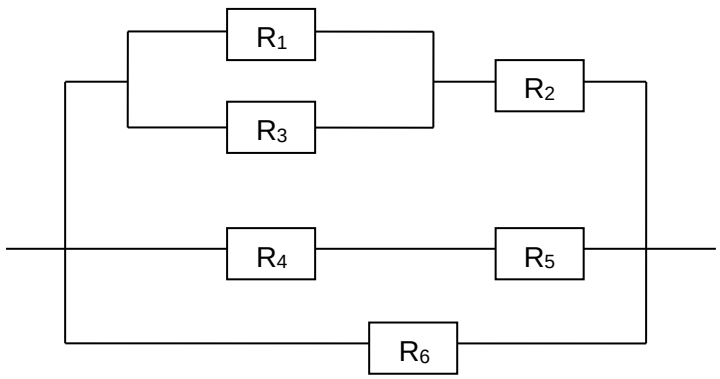
Beispiel: (Widerstandsnetz$_1$ | Widerstandsnetz$_2$ | Widerstandsnetz$_3$)

Achtung! Um mögliche Rundungsfehler zu minimieren, sollten Sie in geeigneter Reihenfolge die Werte verknüpfen bzw. an der "entscheidenden Stelle" beginnend mit dem jeweils kleinesten zum größten Wert hin in aufsteigender Reihenfolge aufaddieren. Die ursprüngliche im Konstruktor übergebene Reihenfolge darf jedoch nicht verloren gehen.

- Sowohl serielle Widerstandsnetze (**SeriesResistor**) wie auch parallele Widerstandsnetze (**ParallelResistor**) sind zusammengesetzte Widerstandsnetze (**ComposedResistor**). Die Konstruktoren mit denen Objekte zusammengesetzter Widerstandsnetze erzeugt werden können, müssen beliebig viele Teil-Widerstandsnetze (**ResistanceNet**) aus denen sich die zusammengesetzten Widerstandsnetze zusammensetzen T als Argumente akzeptieren (Tipp: "varargs-Parameter") und speichern diese.
  Beispiel: **`new ParallelResistor(r1, r3)`**

- Schreiben Sie eine Anwendung, die das nachfolgend abgebildet Widerstandsnetz aufbaut.

Abbildung1: Widerstandsnetz



Die Widerstände R_1 bis R_6 haben die Werte 100 Ω, 200 Ω, ..., 600 Ω.

Für das links abgebildete Widerstandsnetz gilt dann:

- Der Gesamtwiderstand beträgt:
  (etwa) 155.91 Ω
- **`getNumberOfResistors()`** liefert:
  6 Widerstände
- **`getCircuit()`** liefert z.B. den String:
  `(((R1|R3)+R2)|(R4+R5)|R6)`
  Achtung! Die Anzahl Klammern hängt vom Netzaufbau ab.

- Es gibt 2 Arten von Widerständen: Den **OrdinaryResistor** und das **Potentiometer**. Während der **OrdinaryResitor** ein unveränderlicher Widerstand mit konstantem Widerstandswert ist, hat das **Potentiometer** einen regelbaren/veränderlichen Widerstandswert (spezielle Eigenschaft) und daher eine Setter-Methode **`setResistance( double )`** für den zugehörigen Widerstandswert.

- Ersetzen Sie in der oben skizzierten Schaltung den Widerstand R_4 durch ein Potenziometer. Schreiben Sie eine neue Anwendung, die eine Liste der Widerstandswerte der modifizierten Schaltung ausgibt, wenn das Potenziometer in Schritten von 400 von 0 bis auf 4000 Ohm hochgeregelt wird.

- Legen Sie die Referenztypen: **ComposedResistor**, **OrdinaryResistor**, **ParallelResistor**, **Potentiometer**, **ResistanceNet**, **Resistor** und **SeriesResistor** (sowie mögliche sinnvolle(!) Hilfs-Referenztypen, die Sie bei der Implementierung unterstützen) in einem Sub-Package **component** ab.
  Ihr Test bzw. die "Anwendung der Komponenten" muss sich außerhalb dieses (Sub-)Packages befinden.
  Besser wäre, wenn Sie von Anfang an (also schon bei der Entwicklung) mit dem (Sub-)Package **component** arbeiten, weil Sie dann von Anfang an in der Situation sind, für die Sie auch entwickeln.
  Zur Not könnten Sie aber auch Ihre Lösung zunächst ohne das (Sub-)Package implementieren und dann das (Sub-)Package **component** anlegen und dann nur kurz Ihre Lösung in das (Sub-)Package "schieben". Abschließend sollten Sie dann aber noch einmal die Zugriffsrechte auf Sinnhaftigkeit kontrollieren.