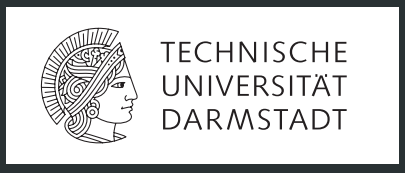
Funktionale und objektorientierte Programmierkonzepte
Übungsblatt 07



Prof. Karsten Weihe

Wintersemester 23/24	v1.1
Themen:	Funktionale Interfaces und Lambda-Ausdrücke
Relevante Foliensätze:	04b und 04c
Abgabe der Hausübung:	15.12.2023 bis 23:50 Uhr

Hausübung 07 — Gesamt: 32 Punkte

Ausdrucksbaum

Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.

Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/h07` und ggf. `src/test/java/h07`.

Verwenden Sie in Ihrem Quelltext 1:1 die auf diesem Übungsblatt gewählten Identifier! Andernfalls wird die jeweilige Aufgabe nicht automatisiert bewertet.

Einleitung

Im Rahmen dieser Übung werden Sie einen *Ausdrucksbaum* implementieren.

Ein Ausdrucksbaum ist eine hierarchische Datenstruktur, die zur Darstellung und Auswertung von Ausdrücken verwendet wird. Er besteht aus Knoten, die verschiedenen Operatoren oder Operanden repräsentieren, und Kanten, die die Beziehungen zwischen den Knoten darstellen. Ein Vorteil liegt in der strukturierten Darstellung der Ausdrücke, bei der die Reihenfolge der Ausführung klar definiert ist. Operanden werden in Blattknoten platziert, während Operatoren in inneren Knoten zu finden sind.

Ein häufiger Anwendungsfall sind *algebraische Ausdrucksbäume*, welche auf Zahlen operieren. In Abbildung 1 ist ein Ausdrucksbaum dargestellt, der die Funktion $e^2 \cdot \frac{x}{\sin(x)}$ repräsentiert.

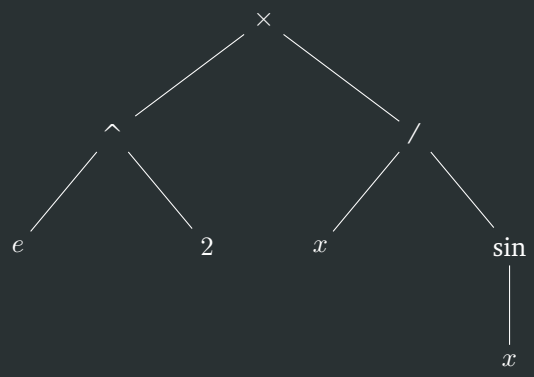


Abbildung 1: Beispiel eines algebraischen Ausdrucksbaums

Die *Auswertung* eines Ausdrucksbaums bezieht sich auf den Prozess, bei dem der Wert eines Ausdrucks durch Traversierung des Baums bestimmt wird. Die Auswertung beginnt bei der Wurzel und arbeitet sich rekursiv nach unten. Das bedeutet, das ein Operator dafür sorgt, das sein Unterbaum bzw. seine Unterbäume ausgewertet werden. Liegt das Ergebnis vor, kann der Operator seine entsprechende Operation ausführen und sein Ergebnis zurückliefern. Dieser Prozess ist rekursiv definiert und kann fortgesetzt werden, bis die Blattknoten erreicht sind.

Sie werden in dieser Übung keinen algebraischen Ausdrucksbaum implementieren, sondern einen Ausdrucksbaum, der zur Formatierung von Objekten des Typ `String` genutzt werden kann. Dazu werden Sie vier verschiedene Knotenarten umsetzen, welche `ValueNode`, `MapNode`, `ConcatenationNode` bzw. `ConditionNode` heißen. `Value-`, `Map-` und `ConditionNode` können verschiedene Realisierungen haben. Beispielsweise kann ein `ValueNode` immer `Hello World!` liefern oder jedes Mal einen anderen Wert. Zur Umsetzung verschiedener Realisierungen werden Sie *Lambda*-Ausdrücke, die in den Klassen `ValueExpression`, `MapExpression` und `ConditionExpression` umsetzen werden.

H1: Interfaces definieren

Die folgenden drei Teilaufgaben zielen darauf ab, dass Sie die `Functional Interfaces` der Ausdrücke implementieren. Alle folgenden Teilaufgaben sind im Package `h07.expression` zu erfüllen.

H1.1: ValueExpression — 1 Punkt

Erstellen Sie in einer Datei `ValueExpression.java` ein `public`-Interface `ValueExpression`, das eine parameterlose Methode `get` mit Rückgabetyp `String` besitzt.

H1.2: MapExpression — 1 Punkt

Erstellen Sie in einer Datei `MapExpression.java` ein `public`-Interface `MapExpression`, das eine Methode `map` mit einem Parameter von Typ `String` und gleichem Rückgabetyp besitzt.

H1.3: ConditionExpression — 1 Punkt

Erstellen Sie in einer Datei `ConditionExpression.java` ein `public`-Interface `ConditionExpression`, das eine Methode `check` besitzt. Diese Methode besitzt einen Parameter von Typ `String` und liefert `boolean` zurück.

H2: Interface implementieren

In dieser Aufgabe werden Sie das Interface `MapExpression` implementieren und lernen wofür sich Lambda-Ausdrücke eignen.

H2.1: ToUpperFormatter — 1 Punkt

Schreiben Sie eine `public`-Klasse `ToUpperFormatter` in eine Datei `ToUpperFormatter.java` im Package `h07.expression.impl`. Die Klasse soll das Interface `MapExpression` implementieren. Die `map`-Methode soll so implementiert werden, dass die Methode `toUpperCase()` des übergebenen `String`-Objekts aufgerufen wird und dessen Rückgabe zurückgegeben wird.

H2.2: Testen — 3 Punkte

In dieser Aufgabe werden Sie Ihre Implementierung testen. Alle in dieser Teilaufgabe erforderten Änderungen sind in Datei `Main.java` in Package `h07` zu erfüllen. Das Gerüst aller zu implementierenden Methoden ist bereits gegeben und muss lediglich wieder einkommentiert werden.

Die Methode `testNormal` soll eine neue Instanz von `ToUpperFormatter` zurückliefern.

`testAnonymous` liefert eine `MapExpression` zurück. Diese soll die Formatierung eines `String`s identisch zu `ToUpperFormatter` erzeugen. Liefern Sie hierbei jedoch kein Objekt von `ToUpperFormatter` zurück, sondern

definieren und initialisieren Sie diese direkt als *anonyme Klasse*. Anonyme Klassen sind namenlose Klassen, die direkt an ihrer Verwendungsstelle definiert und instanziiert werden.

Unbewertete Verständnisfragen:

Wie könnten anonyme Klassen dazu dienen, kurzlebige Implementierungen von Schnittstellen oder abstrakten Typen kompakt und leserlich direkt im Code zu erstellen?

In welchen Situationen erweisen sich anonyme Klassen als besonders praktisch? Fallen Ihnen Situationen ein in denen Klassen nur einmal benötigt werden?

Methode `testLambda` soll, analog zu `testAnonymous`, auch eine `MapExpression` zurückliefern, die sie selbe Formatierung vornimmt. Hier soll dies allerdings als Lambda-Ausdruck realisiert werden. Lambda-Ausdrücke sind gegenüber anonymen Klassen oft bevorzugt, da sie eine kompaktere Syntax bieten. Durch ihre kurze Form verbessern sie die Lesbarkeit des Codes und fördern einen funktionalen Programmierstil.

Auch Methode `testMethodReference` liefert eine `MapExpression` mit gleicher Funktionalität zurück. Sie sollen hier allerdings keinen expliziten Lambda-Ausdruck verwenden, sondern eine sogenannte *Methoden-Referenz* nutzen. Methoden-Referenzen können genau dann genutzt werden, wenn der Lambda-Ausdruck nur aus dem Aufruf einer anderen Methode besteht. In diesem Fall kann unmittelbar diese Methode zurückgegeben werden. Recherchieren Sie in der Java Dokumentation (`https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html`) wie sie Methoden-Referenzen realisieren können und implementieren Sie `testMethodReference` dementsprechend.

Kommentieren Sie anschließend die auskommentierten Ausdrücke aus Methode `test_h22` der Klasse `Main` wieder ein. Sie können nun die `main`-Methode in der selben Datei ausführen und durch die Konsolenausgabe prüfen, ob Ihre Implementierung der drei Methoden korrekt funktioniert. Der String "`FOP for president!`" wird hierbei separat für die drei Implementierung durch die `MapExpression` ausgewertet und auf der Konsole ausgegeben.

H3: Ausdrucksbaum

In dieser Aufgabe werden Sie nun den eigentlichen Ausdrucksbaums implementieren. Alle folgenden Teilaufgaben sind in Package `h07.tree` zu bearbeiten.

H3.1: `Node` 1 Punkt

Schreiben Sie ein `public`-Interface `Node` in einer Datei `Node.java`. Diese Interface wird genutzt um alle Knotentypen zu abstrahieren. `Node` besitzt nur eine Methode `evaluate`, die keinen Parameter besitzt, aber `String` zurückliefert.

H3.2: `ConcatenationNode` 2 Punkte

Schreiben Sie in einer Datei `ConcatenationNode.java` eine `public`-Klasse `ConcatenationNode`, die `Node` implementiert. Der `public`-Konstruktor von `ConcatenationNode` soll zwei Parameter, `left` und `right`, von Typ `Node` besitzen. Diese `Node`-Objekte stellen den Wurzelknoten des linken bzw. rechten Subbaums des aktuellen `ConcatenationNode` dar. Die übergebenen Knoten-Objekte speichern Sie in geeigneten Objektkonstanten ab. Implementieren Sie `evaluate` so, dass der linke und rechte Unterbaum mittels `evaluate` ausgewertet wird und anschließend konkateniert wird.

H3.3: `ValueNode` — 3 Punkte

Sie werden `ValueNode` verwenden, um `ValueExpressions` im Ausdrucksbaum zu verwenden. Schreiben Sie dazu in einer Datei `ValueNode.java` eine `public`-Klasse `ValueNode`, die `Node` implementiert.

Die `ValueNodes` stellen die Blätter des Ausdrucksbaums dar, weshalb diese Knotenart keine Nachfolger besitzt.

Ein `ValueNode` besitzt ein `private`-Attribut `expression` von Typ `ValueExpression`. Initialisieren Sie dieses mit Hilfe eines Lambda-Ausdrucks, sodass ein leerer `String` zurückgeliefert wird. Ein Aufruf von `evaluate` soll den Rückgabewert des `get`-Aufrufs der `expression` weiterleiten.

Des Weiteren besitzt `ValueNode` eine rückgabelose `public`-Methode `setValueExpression` mit einem Parameter von Typ `ValueExpression`. Die übergebene `ValueExpression` soll, wie der Name vermuten lässt, in `expression` gespeichert werden.

H3.4: `MapNode` — 3 Punkte

In dieser Teilaufgabe implementieren Sie `MapNode` in einer `public`-Klasse `MapNode` in einer Datei `MapNode.java` analog zur vorgehenden Teilaufgabe H3.3 den `ValueNode`.

Ein `MapNode` realisiert die Funktionalität einer `MapExpression`. Erstellen Sie daher in `MapNode` ein `private`-Attribut `mapExpression` von Typ `MapExpression`, welches Sie wieder mittels eines Lambda-Ausdrucks initialisieren. Initial soll `mapExpression` einen `String` auf sich selbst abbilden, also die Identitätsfunktion sein. Erstellen Sie außerdem eine rückgabelose `public`-Methode `setMapExpression` mit einem Parameter des formalen Typs `MapExpression`, mit der das Attribut überschrieben werden kann.

Da `MapExpression` ein *unärer* Operator ist, also genau ein Eingangswert benötigt, muss dieser auch im Baum spezifiziert sein. Dazu besitzt der `public`-Konstruktor einen Parameter von Typ `Node`, der in einer geeigneten Objektkonstante gespeichert werden soll.

Implementieren Sie außerdem das Interface `Node` in `MapNode`. Die `evaluate`-Methode soll hierbei den Unterbaum mittels eines Aufrufs von `evaluate` auswerten. Das Ergebnis dieser Auswertung soll durch einen Aufruf von `mapExpression` transformiert und anschließend zurückgegeben werden.

H3.5: `ConditionNode` — 3 Punkte

Sie sollten nun durch die vorgehenden Aufgaben wissen, wie ein Knoten implementiert werden muss. Schreiben Sie daher nach dem gegebenen Muster in einer Datei `ConditionNode.java` eine `public`-Klasse `ConditionNode`, die eine Bedingung mittels `ConditionExpression` umsetzt.

Der `public`-Konstruktor besitzt drei Parameter des formalen Typs `Node`. Der erste `Node` bezeichnet den Unterbaum, der durch die `ConditionExpression` getestet wird, der zweite bzw. dritte Knoten sind die Unterbäume deren Rückgabe der Auswertung zurückgeliefert wird, wenn die `ConditionExpression` `true` bzw. `false` liefert.

Das `private`-Attribut `conditionExpression` von Typ `ConditionExpression` soll mit einem Lambda-Ausdruck initialisiert werden, der immer `false` zurückliefert. Die Methode zum Überschreiben des Attributs gestalten Sie analog zu den vorherigen Aufgaben. Achten sie auf die Einhaltung der Namenskonvention, also `setConditionExpression`.

H4: Logging Engine

In dieser Aufgabe werden wir die Realisierung des Ausdrucksbaums nutzen, um eine Logging Engine zu schreiben.

H4.1: Lambda-Ausdrücke in Methoden erzeugen — 1 Punkt

In dieser Teilaufgabe werden Sie sehen, dass es möglich ist Lambda-Ausdrücke in Methoden zu erzeugen. Wir werden diesen Ansatz nutzen um `MapExpressions` zu erzeugen, die einen `String` in unterschiedlichen Farben einfärben können.

Zur Färbung von `Strings` werden wir *ANSI-Escapesequenzen*[1] nutzen. Escapesequenzen werden Zeichenkombinationen genannt, die nicht als normaler Text, sondern als Sonderfunktion interpretiert werden. Mögliche Funktionen sind zum Beispiel die Positionierung eines Cursors, das Ändern der Schriftgröße oder eben das Einstellen einer Farbe. Das ANSI-Escape ist durch das ESC-Symbol bzw. in Hexadezimal `0x1B` definiert. Möchte man eine Sonderfunktion ausführen, so muss das Escape-Symbol direkt vor der Funktion in der Zeichenkette stehen. Zum Färben stehen nach der originalen Spezifikationen acht Farben für den Vorder- und acht Farben für den Hintergrund bereit. Im Rahmen dieser Übung wollen wir nur die Textfarbe ändern. Die acht Operationen zum Ändern der Textfarbe sind `30` bis `37`. Außerdem gibt es noch Operation `00`, die alle Attribute zurücksetzt. Das Setzen einer Farbe geschieht durch eine Sequenz nach dem Muster: "ESC[<*arg*>m", wobei <*arg*> gegen eine Farbe ausgetauscht werden muss.

Hier ist ein Beispiel:

ANSI-Color Example

```
1  System.out.println("\u001B[31mThis\u001B[0m \u001B[32mis\u001B[0m "
2      + "\u001B[34ma\u001B[0m \u001B[36 mmulti-colored\u001B[0m "
3      + "\u001B[35mexample.\u001B[0m");
```

Dieser Code gibt den Text "This is a multi-colored example." aus. Wir betrachten mal die Escape-Sequenz `"\u001B[31m"`:

Wie man sieht, wird die Farbe `31` gesetzt, was der Farbe rot entspricht.

Diese Aufgabe ist in Datei `Log.java` in Package `h07` zu erfüllen. In der Klasse `Log` sind bereits die Klassenkonstanten `ANSI_BLUE` für blau, `ANSI_YELLOW` für gelb, `ANSI_RED` für rot und `ANSI_RESET` zum Deaktivieren der Farbe definiert, die Sie in Ihrer Implementierung verwenden dürfen.

Passen Sie dazu die bereits definierte Methode `createColorExpression`, die einen Parameter von Typ `String` nimmt, der die ANSI-Escapesequenz einer Farbe repräsentiert, so an, dass eine `MapExpression` zurückgeliefert wird, die einen `String` in die übergebene Farbe einfärbt und die Farbe am Ende des `Strings` wieder zurücksetzt.

Verbindliche Anforderung:

Die Methode `createColorExpression` darf nur aus einer Anweisung, genau der `return`-Anweisung, bestehen.

[1] `https://de.wikipedia.org/wiki/ANSI-Escapesequenz`

H4.2: Formatierung anwenden — 1 Punkt

Passen Sie die bereits definierte `private`-Methode `format` so an, dass die übergebene `message` bzw. `level` in der jeweilig gleichnamigen Objektvariable gespeichert werden. Nutzen Sie den bereits definierten `rootNode`, um ein `String` zurückzuliefern.

H4.3: Formatierung realisieren 1 — 4 Punkte

In Abbildung 2 ist der Ausdrucksbaum gegeben, den Sie in dieser Aufgabe realisieren sollen. Die Spezifikationen des Ausdrucksbaums lauten:

- Der Anfang eines Log-Eintrags besteht aus der aktuellen Zeit, die mittels `LocalTime.now()` ermittelt wird.
- Anschließend kommt ein Trennzeichen ”`:`” und ein Leerzeichen.
- Danach kommt die Nachricht des Textes, welche blau gefärbt wird, wenn die Nachricht Level 0 oder 1 besitzt, gelb, wenn sie Level 2 oder 3 besitzt, und andernfalls rot.
- Die Zeilenumbrüche der Nachricht werden durch das Trennzeichen ”`;`” ersetzt.

In Tabelle 1 sind verschiedene Beispiele dargestellt, damit Sie die Regeln daran nachvollziehen können:

Level	Message	Ausgabe
0	Dies ist ein Test!	15:36:00.921860500: Dies ist ein Test!
3	Kaffeevorrat schwach	15:37:00.927530990: Kaffeevorrat schwach
6	System ausgefallen	15:38:00.634086125: System ausgefallen
0	A\nB\nC	15:39:00.980371845: A;B;C

Tabelle 1: Beispielausgaben des Ausdrucksbaums

Die Implementierung schreiben Sie in der bereits definierten Klasse `NormalLog` in der Datei `NormalLog.java` in Package `h07`. Klasse `NormalLog` ist von `Log` abgeleitet. Um eine Formatierung für den Log zu definieren, implementieren Sie die abstrakte, parameterlose `public`-Methode `generateTree` der Basisklasse, die einen `Node` zurückliefert. Der Rückgabewert ist genau der Wurzelknoten des Ausdrucksbaums.

Verbindliche Anforderung:

In allen Aufrufe der `set`*-Methoden der Operatorenknoten verwenden Sie zwingend Lambda-Ausdrücke.

Eine Ausnahme besteht in der Verwendung der in Aufgabe H4.1 erstellten Methode `createColorExpression`. Die Rückgabe dieser Methode darf auch als Argument der `set`*-Aufrufe genutzt werden.

Es sind verschiedene Baumstrukturen möglich, die die beschriebene Formatierung erreichen. Halten Sie sich bei der Implementierung an den in Abbildung 2 vorgegeben Aufbau, da das Abweichen zu Abzügen in den Tests führen kann.

Hinweis:

Prüfen Sie Ihre Implementierung mit Hilfe der `Main`-Klasse. In dieser ist eine Methode `test_h4` definiert, dessen Inhalt Sie einkommentieren müssen. Anschließend können Sie die `main`-Methode ausführen und schauen, ob Ihre Implementierung funktioniert.

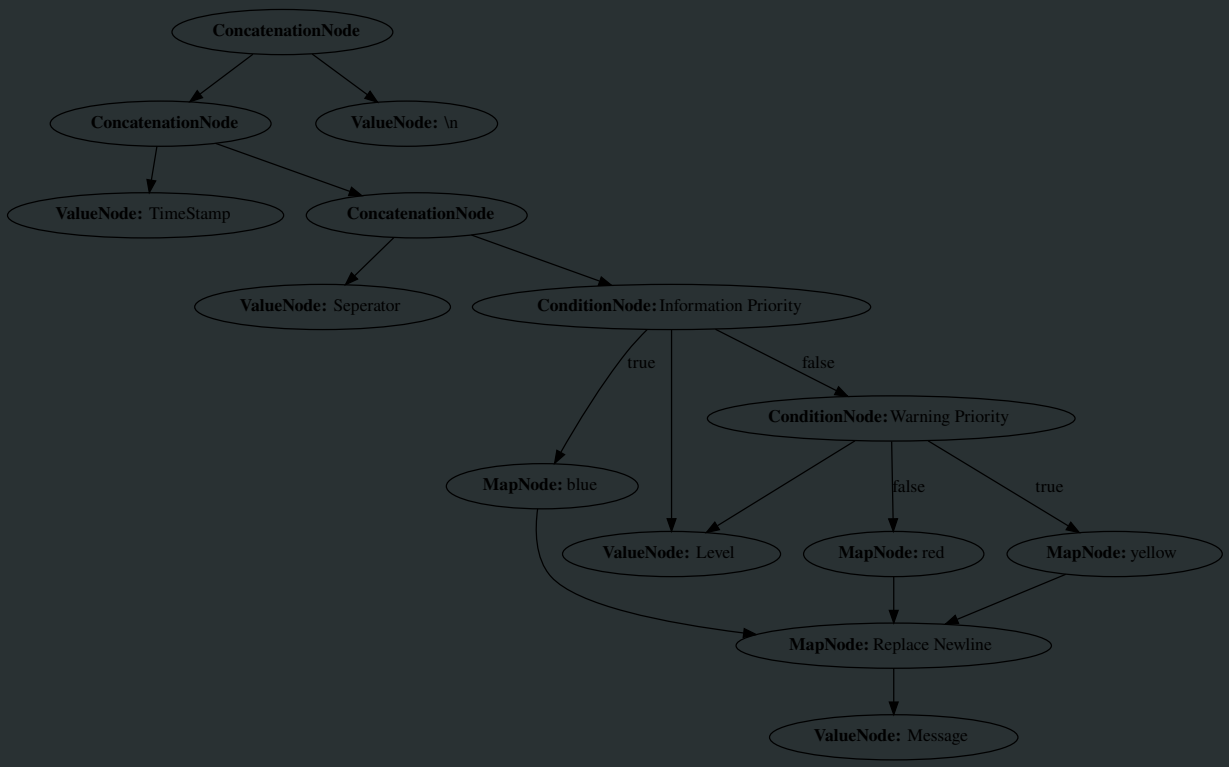


Abbildung 2: Ausdrucksbaum zur Formatierung innerhalb des `NormalLog`

H4.4: Formatierung realisieren 2 — 4 Punkte

In dieser Aufgabe implementieren sie einen `MaintenanceLog` in der Datei `MaintenanceLog.java` in Package `h07`. Klasse `MaintenanceLog` ist von `Log` abgeleitet. Implementieren Sie analog zur vorherigen Teilaufgabe wieder die Methode `generateTree`.

Der `MaintenanceLog` dient Wartungsarbeitern dazu, ausschließlich die für sie relevanten Nachrichten anzuzeigen. Eine Nachricht ist für Maintainer bzw. Wartungsarbeiter genau dann relevant, wenn es sich um eine Nachricht mit Level 3 handelt. Alle anderen Nachrichten sollen komplett herausgefiltert werden, also ein leerer String zurückgeliefert werden. Handelt es sich um eine relevante Nachricht, so soll diese einfach zurückgeliefert werden (mit entsprechendem Zeilenumbruch). Ansonsten müssen keine weiteren Formatierungen beachtet werden.

Verbindliche Anforderung:

In allen Aufrufe der `set*`-Methoden der Operatorenknoten verwenden Sie zwingend Lambda-Ausdrücke.

Hinweis:

Prüfen Sie Ihre Implementierung erneut in der `Main`-Klasse. Passen Sie `test_h4` so an, dass der `MaintenanceLog` verwendet wird.

H5: PowerPlant — 3 Punkte

In dieser Aufgabe werden Sie den in Aufgabe H4 erstellten Logger verwenden, um Ereignisse eines virtuellen Kraftwerks zu loggen. Diese Aufgabe erfüllen Sie ausschließlich in der `public`-Methode `check` in Klasse `PowerPlant` in der Datei

`PowerPlant.java` in Package `h07`. Die Methode hat einen Parameter `t` von Typ `double`, liefert aber nichts zurück. Der Parameter `t` dient hierbei als Zeitvariable.

Eine `PowerPlant` besteht aus mehreren `Reactors`. Alle `Reactors` sind in einem Array `reactors` mit Typ `Reactor` gespeichert.

`Reactor` stellt verschiedene Methoden bereit, um den aktuellen Status zu einem bestimmten Zeitpunkt `t` abzufragen, welche im Folgenden eingeführt werden. Die parameterlose Methode `toString` liefert die Kennung des Reaktors als `String` zurück. Die Methode `getPower` liefert den aktuellen normierten Leistungsausgang des Reaktors als `double` zu einem bestimmten Zeitpunkt `t` zurück, der als `double` übergeben wird. Analog liefert `needMaintenance` für einen Zeitpunkt `t` ein `boolean` zurück, der angibt, ob der Reaktor eine Wartung benötigt.

Als Nächstes implementieren Sie die Methode `check`. Dabei soll über alle Reaktoren des Kraftwerks iteriert, und dabei jeweils deren Status geprüft werden. Dieser wird dann wie folgt über den `Log log` mittels der Methode `log` ausgeben:

- Auf Level 0 geben Sie die aktuelle Leistung in der Form `"<Kennung>: Power = <Power>"` aus, wobei sie `<Kennung>` und `<Power>` gegen den jeweiligen Wert austauschen.
- Wenn die aktuelle Leistung des Reaktors größer als 0.75 ist, geben Sie zudem `"<Kennung>: Overpowered!"` mit jeweiliger Kennung als Level-6-Eintrag aus.
- Wenn der Reaktor eine Wartung benötigt, soll die Nachricht `"<Kennung>: Needs maintenance!"` mit der korrekten Kennung als Level-3-Eintrag geloggt werden.

> **Hinweis:**
>
> Prüfen Sie Ihre Implementierung mit Hilfe der `Main`-Klasse. In dieser ist eine Methode `test_h5` definiert, dessen Inhalt Sie einkommentieren müssen. Anschließend können Sie die `main`-Methode ausführen und schauen, ob Ihre Implementierung funktioniert. Testen Sie Ihre Implementierung sowohl mit Hilfe der Logger-Klassen `NormalLog` als auch `MaintenanceLog`.