# Funktionale und objektorientierte Programmierkonzepte
# Übungsblatt 13

TECHNISCHE UNIVERSITÄT DARMSTADT

Entwurf

**Achtung:** Dieses Dokument ist ein Entwurf und ist noch nicht zur Bearbeitung/Abgabe freigegeben. Es kann zu Änderungen kommen, die für die Abgabe relevant sind. Es ist möglich, dass sich **alle** Aufgaben noch grundlegend ändern. Es gibt keine Garantie, dass die Aufgaben auch in der endgültigen Version überhaupt noch vorkommen und es wird keine Rücksicht auf bereits abgegebene Lösungen genommen, die nicht die Vorgaben der endgültigen Version erfüllen.

## Hausübung 13

**Gesamt: 32 Punkte**

*Codecraft*

**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/h13` und ggf. `src/test/java/h13`.

**Verbindliche Anforderung für die gesamte Hausübung:**

In dieser Hausübung sind alle Klassen aus der Java-Standardbibliothek erlaubt.

## Einleitung

Im Jahr 2009 wurde das bahnbrechende Spiel *Minecraft*[1] auf den Markt gebracht. Es öffnete den Spielern eine schier unendliche Welt, ohne dabei übermäßige Ressourcen zu beanspruchen. Dies wurde durch die prozedurale Generierung der Spielwelt ermöglicht. Spiele wie *Minecraft* und *No Man's Sky*[2] setzen dazu auf komplexe Funktionen, deren Ergebnisse als vielfältige Elemente der Spielwelt interpretiert werden.

In dieser Aufgabe widmen wir uns einem prominenten Vertreter solcher Funktionen - dem *Perlin-Noise*[3]. Dieser wurde 1982 von Ken Perlin für den Film *Tron*[4] entwickelt.

Der Perlin-Noise erzeugt Pseudozufallszahlen, deren Generierung von bereits erzeugten Zahlen beeinflusst wird. Die Ergebnisse können beispielsweise als Höhenwerte einer Landschaft interpretiert werden und fließen nahtlos ineinander über, anstatt - im Gegensatz zu herkömmlichen Zufallsgeneratoren - abruptes Rauschen zu erzeugen. Mit Perlin-Noise können unter anderem Texturen erstellt werden, die sowohl Struktur als auch zufällige Elemente vereinen, etwa in Form von Holzmaserungen.

[1] `https://www.minecraft.net/de-de`
[2] `https://www.nomanssky.com/`
[3] `https://de.wikipedia.org/wiki/Perlin-Noise`
[4] `https://de.wikipedia.org/wiki/Tron_(Film)`

Am Ende dieser Aufgabe werden wir ein Programm entwickeln. Dieses Programm nutzt Perlin-Noise und individuelle Konfigurationen, um eine „Welt“ zu generieren und darzustellen.

In der Abbildung 1 ist eine Vorschau der fertigen Anwendung zu sehen. Ihr Ergebnis sollte ähnlich zu der Abbildung sein. Auf der rechten Seite befinden sich die Konfigurationen für die Weltgenerierung. Auf der linken Seite wird die generierte Welt dargestellt.

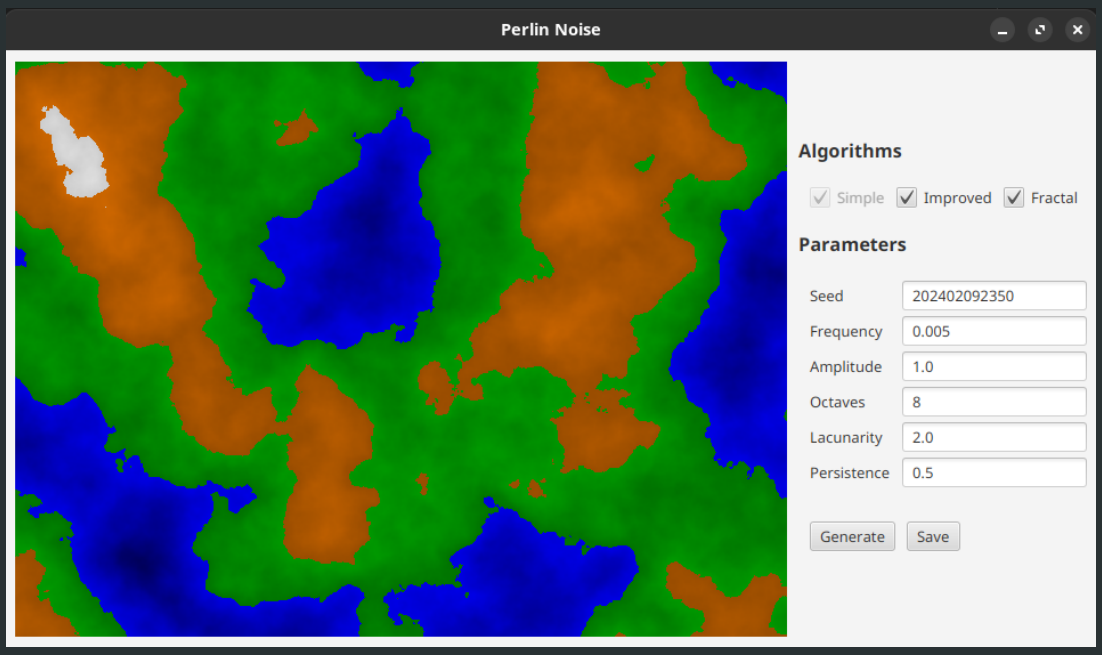



Abbildung 1: Vorschau der fertigen Anwendung auf Linux

## Aufbau

Die Vorlage besteht aus zwei Haupt-Packages - `h13.noise` und `h13.ui`. Das Package `h13.noise` enthält die Implementierung der Perlin-Noise-Algorithmen. Wir werden in diesem Package drei verschiedene Perlin-Noise-Algorithmen implementieren.

Das Package `h13.ui` enthält die Implementierung der Benutzeroberfläche. Diese ist in drei Pakete unterteilt - `h13.ui.controls`, `h13.ui.layout` und `h13.ui.app`.

- `h13.ui.controls`: Dieses Package enthält die Implementierung des benutzerdefinierten `TextField`, die uns nur die Eingabe von Zahlen statt Text ermöglicht.
- `h13.ui.layout`: Dieses Package enthält die einzelnen visuellen Elemente der Benutzeroberfläche. Ein visuelles Element besteht aus einer `View` und einem `ViewModel`. Die `View` ist für die Darstellung der Benutzeroberfläche zuständig und das `ViewModel` für die Logik.
- `h13.ui.app`: Dieses Package enthält die konkrete Implementierung der Anwendung.

## H1: Grundgerüst 7 Punkte

Wir beginnen mit der Erstellung des Grundgerüsts für unseren Algorithmus. Das Interface `PerlinNoise` im Package `h13.noise` stellt eine Funktion dar, die basierend auf Gitterkoordinaten einen Rauschwert erzeugt. Sie dient als Ausgangspunkt für die Implementierung verschiedener Perlin-Noise-Algorithmen.

Die relevanten Klassen befinden sich im Package `h13.noise`.

### H1.1: Gradienten 3 Punkte

Um die Veränderungen der Rauschfunktion zu beschreiben, also wie sich der Rauschwert in benachbarten Bereichen ändert, verwenden wir sogenannte *Gradienten*. Ein Gradient ist in diesem Kontext ein Vektor im Einheitskreis, der dazu dient, die Richtung und Intensität der Veränderungen anzugeben. Diese Gradienten werden in einem Gitter (*Gradient Domain*) gespeichert, das die Welt umspannt (*Noise Domain*). Dies ist der Grund, warum Perlin-Noise als eine auf Gradienten basierende Rauschfunktion betrachtet wird.

In der Abbildung 2 sehen Sie ein Beispiel für eine $3 \times 3$ Welt (*Noise Domain*) mit den zugehörigen Gradienten ($5 \times 5$ *Gradient Domain*).

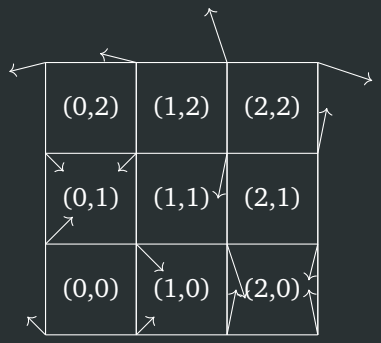



Abbildung 2: $3 \times 3$ Welt mit Gradienten

In der Klasse `AbstractPerlinNoise` implementieren Sie zwei Methoden. Diese helfen Ihnen dabei die Gradienten zu berechnen.

Zuerst müssen wir einen zufälligen Gradienten im Einheitskreis erzeugen. Der Gradient $\vec{g} = \begin{pmatrix} x \\ y \end{pmatrix}$ mit $x, y \in [-1, 1]$ wird als `Point2D` dargestellt. Dafür implementieren Sie die Methode `createGradient()`. Diese Methode gibt einen zufälligen Gradienten zurück. Anschließend können wir mit Hilfe dieser Methode die Gradienten für das Gitter erzeugen. Dies erfolgt in der Methode `createGradients(int, int)`. Diese Methode nimmt als Parameter die *Gradient Domain* entgegen und gibt ein eindimensionalen Array von Gradienten zurück. Die Größe des Arrays entspricht dem Produkt der beiden Parameter. Da wir hier einen zweidimensionalen Array auf einen eindimensionalen Array abbilden, werden die Gradienten breitenweise gespeichert. Beispielsweise wäre das Array (*Gradient Domain*) in einer $2 \times 2$ Welt (*Noise Domain*) folgendermaßen aufgebaut:

$$\{(0,0), (1,0), (2,0), (0,1), (1,1), (1,2), (0,2), (1,2), (2,2)\}$$

Zum Schluss benötigen wir noch eine Funktion $g(x, y)$ mit $x, y \in \mathbb{Z}$, die uns zu einer gegebenen Position in der Welt den zugehörigen Gradienten findet. Dazu implementieren Sie die Methode `getGradient(int, int)`. Diese Methode nimmt als Parameter die *Gradient Domain* entgegen und gibt den zugehörigen Gradienten im Array `gradients` zurück.

**Hinweis:**

(a) Die *Gradient Domain* entspricht der Weltgröße + 2, da wir die Welt mit Gradienten umspannen.

(b) Beachten Sie bei der Implementierung von der Methode `getGradient(int, int)`, dass die Gradienten in einem Gitter gespeichert werden. Das Gitter umspannt dabei die Welt.

**Verbindliche Anforderung:**

Für die zufällige Generierung von Gradienten verwenden Sie das `java.util.Random`-Objekt aus der Klasse `AbstractPerlinNoise`.

## H1.2: Lineare Interpolation und Fading-Funktion 1 Punkt

Für die Berechnung des Rauschwerts benötigen wir eine Funktion $l(x_1, x_2, \alpha)$ mit $x_1, x_2, \alpha \in \mathbb{R}$ (siehe Gleichung 1). Diese Funktion uns ermöglicht uns, zwischen zwei Werten zu interpolieren[5]. $x_1$ und $x_2$ sind die Werte, zwischen denen interpoliert wird. $\alpha$ ist der Interpolationsfaktor, der den Anteil des zweiten Werts $x_2$ am Gesamtergebnis bestimmt.

Implementieren Sie in der Klasse `h13.noise.SimplePerlinNoise` dazu die Methode `interpolate(double, double, double)`. Diese Methode nimmt als Parameter zwei Werte $x_1$ und $x_2$ sowie einen Faktor $\alpha$ entgegen und gibt den interpolierten Wert zurück.

$$l(x_1, x_2, \alpha) = x_1 + \alpha \cdot (x_2 - x_1) \tag{1}$$

Als zweite Funktion für die Berechnung des Rauschwerts benötigen wir eine sogenannte Fading-Funktion $f(t)$ mit $t \in \mathbb{R}$ (siehe Gleichung 2). Hier steht $t$ für den Abstandsfaktor oder die Distanz zum nächsten Gitterpunkt in der Perlin-Noise-Berechnung. Die Funktion $f$ wird verwendet, um den Einfluss der Gradientenvektoren mit zunehmendem Abstand von der Eckposition zu reduzieren. Dieser Verblassungseffekt stellt sicher, dass der Gradienteneffekt näher an der Ecke stärker ist und sich beim Entfernen von der Ecke abschwächt. Das Ergebnis ist eine insgesamt natürlichere und gleichmäßigere Rauschgenerierung.

Implementieren Sie dazu die Methode `fade(double)` in der Klasse `h13.noise.SimplePerlinNoise`. Diese Methode nimmt als Parameter den Abstandsfaktor $t$ entgegen und gibt den Fading-Faktor zurück.

$$f(t) = 6t^5 - 15t^4 + 10t^3 \tag{2}$$

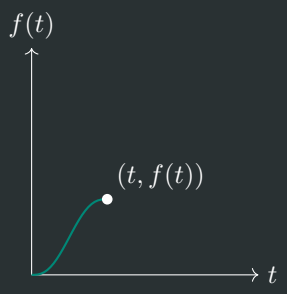



Abbildung 3: Fading-Funktion $f(t)$ zur Reduzierung des Einflusses von Gradientenvektoren mit zunehmendem Abstand.

[5] `https://de.wikipedia.org/wiki/Interpolation_(Mathematik)`

## H1.3: Simpler Perlin-Noise Algorithmus 3 Punkte

Nun haben wir alle Bausteine, um den simplen Perlin-Noise Algorithmus zu implementieren. Implementieren Sie dazu die Methode `compute(double, double)` in der Klasse `SimplePerlinNoise`. Diese Methode nimmt als Parameter die Gitterkoordinaten $(x, y)$ entgegen und gibt den Rauschwert zurück. Gegeben sei also eine Position $(x, y)_{x,y\in\mathbb{R}}$ in einem Gitter. Für die Berechnung des Rauschwertes gehen wir wie folgt vor:

(1) Bestimme $(x_0, y_0)_{x_0,y_0\in\mathbb{Z}}$ und $(x_1, y_1)_{x_1,y_1\in\mathbb{Z}}$ des Gitters, in dem sich $(x, y)$ befindet.

(2) Bestimme die vier Gradienten $\vec{g}_{(x_i,y_j)}$ mit $i, j \in \{0, 1\}$ an den Eckpositionen von $(x, y)$.

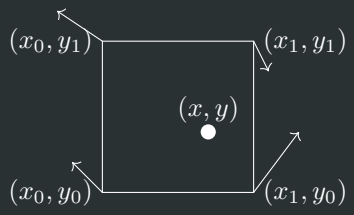



Abbildung 4: Punkt $(x, y)$ und seine zugehörigen Gradienten an den Eckpositionen

(3) Berechne den Distanzvektor $\vec{d} = (\vec{d_x}, \vec{d_y})^T = (x - x_0, y - y_0)^T$ zwischen $(x, y)$ und der Gradienten-Ecke $(x_0, y_0)$.

(4) Dann berechnen wir das Skalarprodukt zwischen dem Distanzvektor und dem Gradientenvektor der jeweiligen Ecke (siehe Gleichung 3).

$$\begin{aligned} s_{(x_0,y_0)} &= (\vec{d} + (0,0)^T) \cdot \vec{g}_{(x_0,y_0)} & s_{(x_1,y_0)} &= (\vec{d} + (-1,0)^T) \cdot \vec{g}_{(x_1,y_0)} \\ s_{(x_0,y_1)} &= (\vec{d} + (0,-1)^T) \cdot \vec{g}_{(x_0,y_1)} & s_{(x_1,y_1)} &= (\vec{d} + (-1,-1)^T) \cdot \vec{g}_{(x_1,y_1)} \end{aligned} \tag{3}$$

(5) Interpoliere die Ergebnisse des Skalarprodukts mit der Interpolationsfunktion. Die Parameter der Interpolationsfunktion sind die Skalarprodukte der gleichen $x$-Koordinate des Distanzvektors mit dem geglätteten $\vec{d_x}$ (siehe Gleichung 4).

$$\begin{aligned} lx_0 &= l(s_{(x_0,y_0)}, s_{(x_1,y_0)}, f(\vec{d_x})) \\ lx_1 &= l(s_{(x_0,y_1)}, s_{(x_1,y_1)}, f(\vec{d_x})) \end{aligned} \tag{4}$$

(6) Schließlich interpolieren wir die beiden interpolierten Werte erneut mit der Interpolationsfunktion. Als $\alpha$ verwenden wir die geglättete $\vec{d_y}$ (siehe Gleichung 5).

$$l = f(l_{x_0}, l_{x_1}, f(\vec{d_y})) \tag{5}$$

**Hinweise:**

(a) Um einen Gradienten an der Position $(x, y)$ zu finden, können Sie die Methode `getGradient(int, int)` verwenden. Diese Methoden haben Sie in der H1.1 implementiert.

(b) Sie können die Methode Point2D#dotProduct(Point2D) verwenden, um das Skalarprodukt zwischen zwei Vektoren zu berechnen.

## H2: Erweiterte Perlin-Noise Algorithmen 3 Punkte

In der H1 haben wir den simplen Perlin-Noise Algorithmus implementiert. In dieser Aufgabe werden wir weitere Perlin-Noise Algorithmen implementieren. Diese sind etwas komplexer als der simple Perlin-Noise Algorithmus.

Die relevanten Klassen befinden sich im Package `h13.noise`.

### H2.1: Permutationstabelle 2 Punkte

Die verbesserte Version von dem simplen Perlin-Noise Algorithmus verwendet eine Permutationstabelle und unterscheidet sich nur bei der Auswahl von Gradienten-Ecken.

Hier werden die Gradienten anhand einer Permutationstabelle $p(i)_{i\in[0,2\cdot256)}$ ausgewählt, die zufällig generiert wird. Die Permutationstabelle ist $2 \cdot 256$ groß und die ersten $256$ sind aufsteigend geordnet. Die zweite Hälfte der Tabelle ist eine Kopie der ersten Hälfte, die zufällig permutiert wurde. Dies ist nur eine von vielen Möglichkeiten, eine Permutationstabelle zu generieren. Einer der klassischen Anordnungen der Permutationstabelle finden Sie auf `https://en.wikipedia.org/wiki/Perlin_noise#Permutation`.

Implementieren Sie in dieser Aufgabe die Methode `createPermutation(Random)`. Die Methode befindet sich in der Klasse `ImprovedPerlinNoise`. Sie nimmt als Parameter ein `java.util.Random`-Objekt entgegen und gibt die Permutationstabelle zurück. Die Anordnung der Elemente in der Permutationstabelle entnehmen Sie aus dem vorherigen Absatz.

Haben Sie dies getan, müssen wir nur noch die Methode `getGradient(int, int)` anpassen, um die Gradienten anhand der Permutationstabelle auszuwählen. Die Auswahl des Gradienten erfolgt anhand folgenden Schema für eine Permutationstabelle der Größe $2 \cdot 256$:

$$\tilde{g}(x, y) = p((x + p(y \ \& \ 255)) \ \& \ 255) \tag{6}$$

Das Ergebnis von $p$ gibt uns den Index der Permutationstabelle zurück, an dem der auszuwählende Gradient gespeichert ist. Das bedeutet $\tilde{g}(x, y)$ gibt uns an, welchen Gradienten wir aus der *Gradient Domain* auswählen müssen.

**Verbindliche Anforderungen:**

(i) Verwenden Sie das `java.util.Random` Objekt von dem übergebenen `PerlinNoise`-Objekt für die zufällige Anordnung der zweiten Hälfte der Permutationstabelle.

(ii) Die Elemente in der Permutationstabelle sind vom Typ `int` und haben den Wertebereich $[0, 255]$.

**Hinweise:**

(a) Die Permutationstabelle $p$ finden Sie in der Klasse `ImprovedPerlinNoise` als Attribut namens `p`.

(b) Die Methode `getGradient()` gibt Ihnen die *Gradient Domain* zurück.

(c) Das kaufmännische Und-Zeichen, also &, stellt das „bitweise Und“ dar.

## H2.2: Fraktale 1 Punkt

Die Geländeformen in der realen Welt sind deutlich rauer. Die bisher erstellte Welt ist deutlicher glatter und erzeugt daher einen unrealistischen Eindruck. Um dies zu erreichen, müssen wir Rauschen hinzufügen. Hier kommen die Fraktale ins Spiel, die wir in der Methode `compute(double, double)` von der Klasse `FractalPerlinNoise` implementieren. Wir starten bei einer Grundfrequenz $f_0$ und Grundamplitude $a_0$, die Sie aus den Attributen bzw. Methoden der Klasse entnehmen können. Die Berechnung des Rauschwertes $N(x, y)$ erfolgt wie folgt:

$$\begin{aligned} N(x,y) &= \sum_{i=0}^{o-1} U(x \cdot f_i, y \cdot f_i) \cdot a_i \\ f_i &= f_{i-1} \cdot l \\ a_i &= a_{i-1} \cdot p, \end{aligned} \tag{7}$$

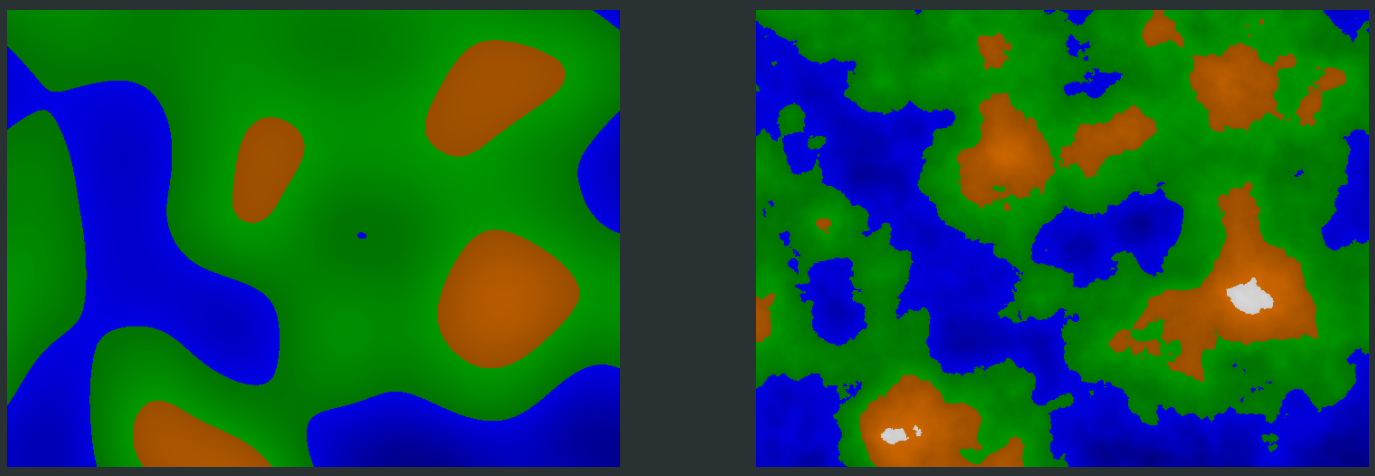
wobei $l$ für `lacunarity`, $p$ für `persistence`, $o$ für `octaves` und $U(x, y)$ für `compute(double, double)` des Basisalgorithmus (Attribut `delegate` [6]) stehen.

**Hinweis:**

Das $\Sigma$ steht für das Summenzeichen[a]. Wir beginnen bei $i = 0$ und gehen bis $i = o - 1$ und summieren die Ergebnisse von $U(x \cdot f_i, y \cdot f_i) \cdot a_i$ auf.

[a] `https://de.wikipedia.org/wiki/Summe#Notation_mit_dem_Summenzeichen`

In der Abbildung 5 können Sie die Unterschiede zwischen dem simplen Perlin-Noise und dem fraktalen Perlin-Noise sehen. Man kann deutlich erkennen, dass der fraktale Perlin-Noise deutlich rauer ist als der simple Perlin-Noise.



(a) Simpler Perlin-Noise

(b) Fraktaler Perlin-Noise

Abbildung 5: Vergleich Perlin-Noise Algorithmus

[6] https://de.wikipedia.org/wiki/Delegation_(Softwareentwicklung)

## H3: Controls 5 Punkte

In JavaFX gibt es Textfelder, die es ermöglichen, Texte einzugeben. In unseren Fall wollen wir aber nur Zahlen eingeben, weshalb wir unser eigenes Textfeld implementieren, das nur Zahlen akzeptiert.

Für diese Aufgabe befinden sich alle Klassen im Package `h13.ui.controls`.

### H3.1: Converter 2 Punkte

Damit unser Textfeld nur Zahlen akzeptiert, müssen wir einen `javafx.util.StringConverter` implementieren, der die Eingabe in eine Zahl umwandelt. Die Klasse `StringConverter` bietet zwei Methoden an, um einen String in Typ `T` bzw. in unseren konkreten Fall `Number` umzuwandeln und umgekehrt.

Implementieren Sie dazu die Klasse `NumberStringConverter` die Methode `toString(Number)`. Diese Methode gibt eine Zahl als einen `String` mittels dem `stringifier` zurück. Falls die Eingabe `null` ist, so wird ein leerer String zurückgegeben.

Haben Sie dies getan, implementieren Sie anschließend die Methode `fromString(String)`. Diese Methode wandelt die einen `String` in eine Zahl mittels dem `numericizer` um. Falls die Eingabe `null` oder leer ist, so wird `null` zurückgegeben. Ansonsten wird der `String` in eine Zahl umgewandelt. Beachten Sie, dass die Eingabe `"-"` als `"-1"` interpretiert wird.

> **Hinweis:**
>
> Beachten Sie, dass die Eingabe von `fromString(String)` Leerzeichen enthalten kann. Entfernen Sie zuerst die Leerzeichen, bevor Sie den `String` in eine Zahl umwandeln. Um die Leerzeichen in einem String zu entfernen, können Sie die Methode `java.lang.String#trim` verwenden.

### H3.2: Nummerfeld 3 Punkte

Mittels dem Konverter aus H3.1 können wir nun unsere eigene Textfelder implementieren, die nur Zahlen akzeptieren. Dafür implementieren Sie die Methode `initBindings()` in der Basisklasse `NumberField`, die ein `TextField` erweitert und die Eingabe auf Zahlen beschränkt.

Die Methode `initBindings()` wird verwendet, um bei einer Änderung des Textfeldes das Attribut `value` zu aktualisieren. Da der Text als String gespeichert wird, existiert das Attribut `value`. Das Attribut `value` ist vom Typ `Property<Number>` und repräsentiert zu jedem Zeitpunkt den Wert des Textfeldes als Zahl. Konkret bedeutet dies, dass das Attribut `value` eine Benachrichtigung erhält, wenn sich der Wert des Textfeldes ändert, und umgekehrt. Die Konvertierung von einem `String` in eine Zahl und umgekehrt erfolgt mittels der abstrakten Methode `getConverter()`.

Zum Schluss können wir in den Klassen `IntegerField`, `LongField` und `DoubleField` die abstrakte Methode `getConverter()` aus der Basisklasse implementieren. Die Methode soll uns einen passenden Konverter für den jeweiligen Typ zurückgeben. Den Typ können Sie aus dem Klassennamen entnehmen. Verwenden Sie hierzu die Klasse `NumberStringConverter` aus H3.1.

> **Hinweis:**
>
> Die Methode `Bindings.bindBidirectional(Property, Property, StringConverter)` könnte für die Implementierung hilfreich sein.

## H4: Konfigurations-Menü 7 Punkte

Bevor wir den Perlin-Noise Algorithmus visualisieren können, benötigen wir eine Möglichkeit, den Algorithmus zu konfigurieren. Beispiele für die Konfiguration sind Parameter und die Art des Algorithmus.

Für diese Aufgabe befinden sich alle Klassen im Package `h13.ui.layout`.

### H4.1: Algorithmusauswahl 1 Punkt

In der Vorlage finden Sie bereits die Klasse `ParameterView`. Diese Klasse ermöglicht einer Person, Parameter für den Perlin-Noise Algorithmus auszuwählen. Wir möchten nun einer Person ermöglichen, den Perlin-Noise Algorithmus auszuwählen. Dazu verwenden wir die Klasse `ChooserView`. Diese Klasse verwendet intern ein `javafx.scene.layout.GridPane` (Attribut `root`) und stellt die Auswahlmöglichkeiten via `CheckBoxen` dar.

Die Auswahlmöglichkeiten werden in einer `javafx.collections.ObservableMap` gespeichert mit jeweils den Namen des Algorithmus als Schlüssel und die `javafx.scene.control.CheckBox` als Wert. Die `Map` repräsentiert zu jedem Zeitpunkt die Auswahlmöglichkeiten in der Benutzeroberfläche, d.h. wenn eine Veränderung in der `Map` stattfindet, soll die Auswahlmöglichkeiten in der Benutzeroberfläche aktualisiert werden und umgekehrt. Implementieren Sie die Methode `initialize()` von der Klasse `ChooserView`. Die Methode aktualisiert die Auswahlmöglichkeiten in der Benutzeroberfläche, falls sich die `Map` ändert. Das heißt, falls ein Element (Wert) aus der `Map` eingefügt oder entfernt wird, soll das Element in `root` eingefügt werden.

**Verbindliche Anforderungen:**

1. Es wird immer eine Reihe in der `GridPane` befüllt, bevor eine neue Reihe angefangen wird. Die Position im `GridPane` wird mittels der Attribute `nextColumn` und `nextRow` bestimmt, wobei unsere `ChooserView` eine fixe Spaltenanzahl (`columnSize`) besitzt. Falls die Spaltenanzahl überschritten wird, soll die nächste Reihe verwendet werden. Sie können sich hierbei an der Klasse `ParameterView` orientieren.
2. Aktualisieren Sie die Attribute `nextColumn` und `nextRow` bei einer Änderung der `Map`.

**Hinweis:**

Eine `ObservableMap` ist eine `Map`, die es ermöglicht, Änderungen an der Map zu beobachten. Mittels der Methode `addListener(MapChangeListener)` können wir eine `MapChangeListener` hinzufügen, die bei einer Änderung der Map aufgerufen wird.

Schauen Sie sich am besten das funktionale Interface `javafx.collections.MapChangeListener` an, das Ihnen hilfreiche Methoden zur Verfügung stellt.

## H4.2: Welche Parameter für welchen Algorithmus? 5 Punkte

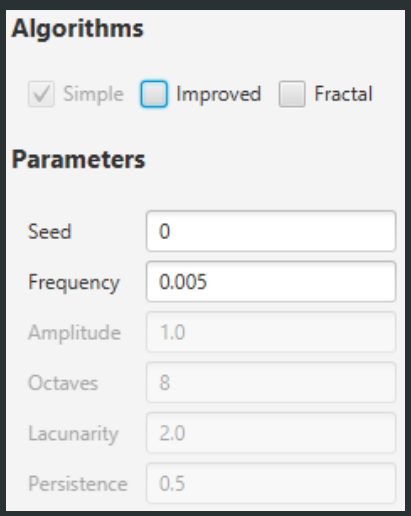
Unser simplen Perlin-Noise Algorithmus benötigt nur einen Startwert (`seed`) und die Frequenz (`frequency`). Der fraktale Perlin-Noise Algorithmus benötigt zusätzlich noch die Amplitude (`amplitude`), die Lacunarity (`lacunarity`) und die Persistence (`persistence`). In der Abbildung 6 können Sie die Unterschiede zwischen den simplen Perlin-Noise und dem fraktalen Perlin-Noise Algorithmus sehen. Die ausgegrauten Parameter stellen die Parameter dar, die nicht verfügbar sind.

Das heißt, wir müssen die Parameterauswahl passend zum ausgewählten Algorithmus verfügbar machen. Dies wird mittels der Methode `addVisibilityListener(Map<String, Set<String>> configurations)` in der Klasse `h13.ui.layout.SettingsViewModel` realisiert. Diese Methoden implementieren Sie nun.

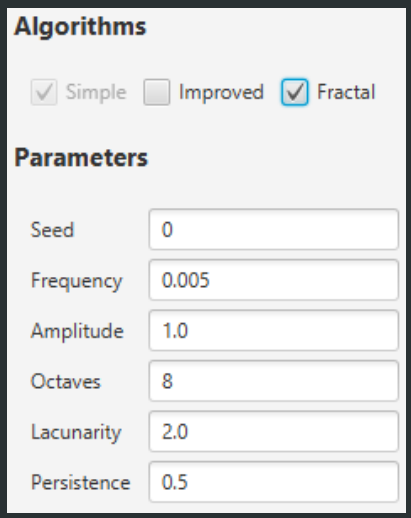
Die Methode erhält als Parameter eine `Map` mit den Namen des Algorithmus als Schlüssel und die Namen der Parameter als Wert, die für den jeweiligen Algorithmus benötigt werden. Die Auswahl eines Algorithmus wird im Attribut `algorithms` gespeichert. Die Sichtbarkeiten der Parameter werden im Attribut `parameters` gespeichert. Die beiden Attribute speichern die Namen der Algorithmen bzw. Parameter als Schlüssel und die Auswahl bzw. Sichtbarkeiten als Wert.

Für die Sichtbarkeit des simplen Perlin-Noise Algorithmus, wie in der Abbildung 6 zusehen, hat die `Map` einen Schlüsselwert von `"Simple"` und den zugehörigen Wert eines `Set` mit den Namen der Parameter `"Seed"` und `"Frequency"`.

Die Aufgabe des Listener ist es einen `javafx.beans.binding.BooleanBinding` für alle Parameter zu erstellen. Dieses gibt an, wann ein Parameter verfügbar bzw. nicht verfügbar sein soll. Haben Sie die `BooleanBinding`s korrekt erstellt, so müssen Sie diese an die Sichtbarkeit Eigenschaft (`BooleanProperty`) der jeweiligen Parameter anbinden.




(a) Simpler Perlin-Noise




(b) Fraktaler Perlin-Noise

Abbildung 6: Vergleich des Perlin-Noise Algorithmus mit benötigen Parametern

**Hinweise:**

(a) Speichern Sie die einzelnen `BooleanBindings` in einer `Map` mit dem Namen des Parameters als Schlüssel, damit Sie die `BooleanBindings` später an die Sichtbarkeit Eigenschaft anbinden können.

(b) Die Methode `Bindings#createBooleanBinding(Callable, Observable...)` ermöglicht es Ihnen, einen `BooleanBinding` zu erstellen, der sich automatisch aktualisiert, falls sich die übergebenen `Observables` ändern. Initialisieren Sie diesen mit einem `BooleanProperty` mit dem Startwert `false`.

(c) Gehen Sie alle Parameter durch und schauen Sie für jeden Algorithmus, ob der Parameter benötigt wird und binden Sie den `BooleanProperty` an den zugehörigen `BooleanBinding` aus Schritt (a).

### H4.3: Konfigurations - Ansicht 1 Punkt

Nun haben wir alle Bausteine, um die Konfigurationen in der Benutzeroberfläche zu integrieren. Dafür implementieren Sie die Methode `initialize()` in der Klasse `h13.ui.layout.SettingsView`. Die Klasse verwendet intern eine `javafx.scene.layout.HBox` (Attribut `root`) und stellt die Konfigurationen mittels der Implementationen aus der Klasse `ParameterView` und `ChooserView` dar.

Die Methode soll zuerst die Algorithmenauswahl (Attribut `algorithms`) und dann die Parameterauswahl (Attribut `parameters`) in die `VBox` einfügen. Hierbei soll zuerst das `Label` und anschließend die Ansicht für die Auswahl eingefügt werden. Zum Schluss soll die Button-Gruppe (`buttonGroup`) eingefügt werden. In die Button-Gruppe müssen Sie die `Button generate` und `save` einfügen. Hier soll der `generate` Button links und der `save` Button rechts eingefügt werden.

Jetzt fehlt nur noch die Sichtbarkeit der Parameter zu den zugehörigen Algorithmen. Rufen Sie dazu die Methode `addVisibilityListener(Map<String, Set<String>> configurations)` von dem Attribut `viewModel` auf und übergeben Sie die `Map` von dem Attribut `visibilities`.

> **Hinweis:**
>
> Schauen Sie sich das Interface `h13.ui.layout.View` an. Das Interface besitzt eine Methode `getView()` mit dem Rückgabetyp `javafx.scene.Parent`, das die Ansicht repräsentiert. Diese Ansicht müssen Sie in die `SettingsView` einfügen.

## H5: Algorithmus-Ansicht 7 Punkte

Jetzt fehlt uns nur noch die Ansicht für die Visualisierung des Perlin-Noise Algorithmus. Alle relevanten Klassen befinden sich im Package `h13.ui.layout`.

### H5.1: P-I-C 3 Punkte

Die Klasse `AlgorithmViewModel` dient als Basisklasse für die Darstellung verschiedener Perlin-Noise-Algorithmen auf einem `Canvas`. Sie stellt Funktionen bereit, um die Welt zu generieren und anzuzeigen. Die konkreten Algorithmen werden in den Unterklassen festgelegt.

Ihre Aufgabe ist es, die Methode `createImage(PerlinNoise, int, int, int, int)` zu implementieren. Diese Methode nimmt einen Algorithmus (`PerlinNoise`) als Eingabe. Sie erzeugt ein Bild von den Koordinaten $(x, y)$ bis $(x + w, y + h)$, basierend auf den Startkoordinaten und der Größe. Beginnen Sie damit, ein `WritableImage` mit den Abmessungen $(w, h)$ zu erstellen. Dieses Bild wird zur Zeichnung verwendet und am Ende zurückgegeben. Um auf dem Bild zu zeichnen, benötigen Sie einen `PixelWriter`. Den `PixelWriter` erhalten Sie über die Methode `getPixelWriter()` von `WritableImage`. Verwenden Sie die Methode `setColor(int, int, Color)`, um die Farbe eines Pixels festzulegen. Verwenden Sie die Methode `compute(int, int)` des Algorithmus, um den Rauschwert zu erhalten. Zeichnen Sie dann die (Teil-)Welt und verwenden Sie die `colorMapper`-Funktion, um aus einem Rauschwert die entsprechende Farbe zu generieren.

Die Methode `draw` verwendet den übergebenen Algorithmus und zeichnet diesen auf dem `Canvas` über den `GraphicsContext`. Falls der Algorithmus `null` ist, gibt es nichts zu tun. Wenn der Algorithmus nicht normalisiert ist, normalisieren Sie ihn, indem Sie die Klassenmthode `normalized(PerlinNoise)` von `h13.noise.PerlinNoise` aufrufen. Speichern Sie nach jedem Wechsel des Algorithmus den aktuellen Algorithmus in der Variable `lastAlgorithm` und zeichnen Sie die (Teil-)Welt mit der Methode `drawImage` von `GraphicsContext`.

**Unbewertete Verständnisfrage:**

Wir haben für die Parameterwerte nur eine untere Schranke definiert. Je nachdem, wie wir die Parameter wählen, könnte es bei der Berechnung und Konvertierung des Rauschwertes vorkommen, dass die Werte außerhalb des Farbraums liegen.

Was würde in diesem Fall passieren und wie könnte man dies der nutzenden Person mitteilen? Sie müssen dies nicht implementieren, können dies aber gerne tun. Dies würde aber nicht bewertet werden. Achten Sie darauf, dass die Implementierung trotzdem für valide Parameterwerte funktionieren muss.

## H5.2: Zoom in, zoom out — 4 Punkte

Wir haben nun eine Möglichkeit ein Bild zu generieren bzw. speichern und darzustellen. Was uns jetzt fehlt, ist diese Aktion an den `Button` zu binden. Damit der `Button` den aktuell ausgewählten Algorithmus (`getAlgorithm()`) auf der kompletten Größe des `Canvas` zeichnet. Der ausgewählte Algorithmus stammt aus der Klasse `h13.ui.layout.AlgorithmViewModel` stammt.

Implementieren Sie dazu die Methode `initializeButtons()` und `initializeSize()`. Beide Methoden befinden sich in der Klasse `h13.ui.layout AlgorithmView`.

Die Methode `initializeButtons()` soll die Aktionen beim Drücken der `Buttons` `generate` und `save` festlegen.

Beim Drücken des `Buttons` `generate` soll das Bild mittels der Methode des ViewModels `draw(PerlinNoise, GraphicsContext, int, int, int, int)` auf die Zeichenfläche zeichnen. Wählen Sie hierzu den aktuellen Algorithmus aus und zeichnen Sie über die komplette verfügbare Fläche des `Canvas`. Falls der `Button` `save` gedrückt wird, soll das Bild mittels der Methode `save(int, int)` mit der Größe der Zeichenfläche gespeichert werden.

Zum Schluss müssen wir noch angeben, was passieren soll, wenn sich die Größe des `Canvas` ändert bzw. des `BorderPanes`. Die Zeichenfläche ist immer so groß wie die verfügbare Fläche des `BorderPanes` minus die Fläche für die Konfigurationen. Falls sich die Zeichenfläche ändert, soll mittels der Methode `draw(PerlinNoise, GraphicsContext, int, int, int, int)` die **neue** Zeichenfläche gezeichnet werden.

**Hinweise:**

(a) Die `Buttons` `generate` und `save` erhalten Sie mittels der Getter-Methoden aus dem `SettingsView`.

(b) Die `GraphicsContext2D` erhalten Sie mittels der Methode `getGraphicsContext2D` von Canvas.

(c) Beachten Sie bei der Breite die `SettingsView` und bei der Höhe das `Padding` des `BorderPanes`. Erstellen Sie bspw. `BooleanProperty` für das obere und untere `Padding` und binden Sie diese an die `BorderPane`, damit die beiden Variablen die Paddings der Ansicht zu jedem Zeitpunkt repräsentieren. Diese können Sie dann bei der Bestimmung der neuen Höhe abziehen.

## H6: App 3 Punkte

Die Anwendung ist fast abgeschlossen. Es fehlt nur noch die Implementierung der Methode `getAlgorithm`. Diese Methode implementieren Sie in der Klasse `h13.ui.app.PerlinNoiseViewModel`. Die Klasse unterstützt drei Perlin-Noise Algorithmen - den simplen, den verbesserten und den fraktalen Algorithmus.

Für alle Algorithmen benötigen wir den Startwert (`Algorithmn.SEED`) und die Frequenz (`Algorithm.FREQUENCY`). Damit wir nicht jedes Mal die Gradienten für einen gegebenen `seed` (Startwert für eines `Random`-Objektes) berechnen müssen, speichern wir für den simplen und den verbesserten Algorithmus diese jeweils ab (`cacheSimpleNoise` und `cacheImprovedNoise`).

Zunächst überprüfen wir den `cacheSimpleNoise`, um festzustellen, ob bereits ein Algorithmus mit dem gleichen `seed` gespeichert ist. Wenn sich kein Algorithmus im Cache befindet, erstellen wir einen neuen Algorithmus. Wir speichern den Algorithmus im Cache, zusammen mit den aktuellen Display-Abmessungen, der Frequenz und dem Startwert für das `Random`-Objekt.

Als nächstes prüfen wir, ob der verbesserte Algorithmus ausgewählt wurde. Die `cacheImprovedNoise` funktioniert analog zu `cacheSimpleNoise`. Als Unterschied verwenden wir hier als Schlüsselwert den simplen Algorithmus statt den `seed`. Analog zum simplen Algorithmus erstellen wir einen neuen Algorithmus und speichern ihn im Cache, falls kein Algorithmus im Cache vorhanden ist. Verwenden Sie bei der Erstellung des verbesserten Algorithmus den simplen Algorithmus aus dem Cache und die Permutationstabelle via `getPermutationTable(Random)`.

Zum Schluss überprüfen wir, ob der fraktale Algorithmus ausgewählt wurde. Falls ja, erstellen wir einen neuen fraktalen Algorithmus mit den angegebenen Parametern: `amplitude`, `octaves`, `Lacunarity` und `persistence`.

Wenn der ausgewählte Algorithmus dem zuletzt ausgeführten Algorithmus entspricht und die Frequenz gleich ist, geben wir `null` zurück. Auf diese Weise zeichen wir keinen neuen Algorithmus, da sich der Algorithmus nicht verändert hat. Andernfalls aktualisieren wir den zuletzt ausgeführten Algorithmus auf den aktuellen Algorithmus und setzen die Frequenz entsprechend.

Haben Sie dies getan, so können Sie die Anwendung ausführen (Klasse `Main`) und die verschiedenen Algorithmen ausprobieren. Die Anwendung können Sie mittels der Gradle-Task `application/run` ausführen.

**Hinweise:**

(a) Um einen Parameter zu erhalten, können Sie die Methode `getParameter(Parameter)` nutzen. Die Parameternamen sind bereits in der Enumeration `Parameter` definiert. Die Methode gibt Ihnen den Parameter als `Property<Number>` zurück.

(b) Um einen Algorithmus zu erhalten, können Sie die Methode `getAlgorithm(Algorithm)` nutzen. Die Algorithmenamen sind bereits in der Enumeration `Algorithm` definiert. Die Methode gibt Ihnen den Algorithmus als `BooleanProperty` zurück, d.h. wenn der Algorithmus ausgewählt ist, so ist der Wert `true`, ansonsten `false`.

(c) Die Display-Abmessungen können über `Screen.getPrimary().getVisualBounds()` abgerufen werden.