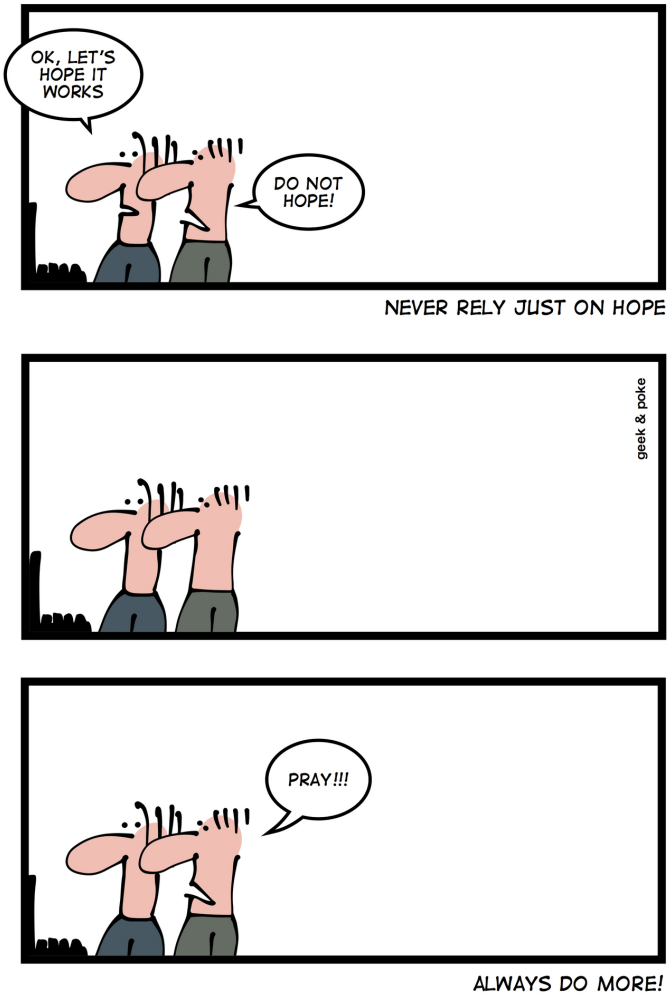
# Universiteit van Amsterdam

## Opgave 7

Jordy Perlee – Stephen Swatman – Robin de Vries – Engel Hamer

# Lindenmayer




**Deadline: zaterdag 19 oktober 2019, 18:00**

# Voorwoord

Voor je ligt de laatste opgave van Inleiding Programmeren. Echt makkelijk is hij niet, maar na de vorige zes opgaven moet het te doen zijn. We hopen dat je de afgelopen weken veel hebt geleerd van je tripje door de wondere wereld van programmeren in Java en dat je er natuurlijk ook heel veel plezier aan hebt beleefd. Succes met de laatste opdracht!

# 1 Introductie

Tot nu toe was iedere week het programmeren van een applicatie het einddoel. Je moest applicaties maken die je (aan de hand van invoer van een gebruiker) een bepaalde uitvoer gaf, denk bijvoorbeeld aan de hotel-opgave van vorige week. Uiteraard is programmeren vaak niet het doel zelf, maar het middel. Eén van de belangrijkste subdisciplines van de Informatica is het modelleren en simuleren. Bij het modelleren en simuleren wordt een applicatie gemaakt die –aan de hand van een aantal vooraf opgestelde regels– een proces uit de echte wereld zo nauwkeurig mogelijk afbeeldt. Het is een belangrijk onderdeel van de informatica omdat het ons leert om natuurlijke processen uit de echte wereld beter te begrijpen.

In deze opdracht ga je zelf een kleine simulatie implementeren om kennis te maken met het modelleren en simuleren. De simulatie die je gaat maken implementeert een Lindenmayer-systeem[1] (ofwel: L-systeem). Dit systeem transformeert een zogeheten *axioma* (axiom), bestaande uit een set van symbolen (een string in Java terminologie) uit een vooraf gedefinieerd alfabet. Lindenmayer-systemen waren in eerste instantie ontwikkeld om de groei van planten te kunnen beschrijven en modelleren. De transformaties in een Lindenmayer-systeem worden gedaan aan de hand van een set van regels welke ook vooraf worden meegegeven. Het resultaat van deze transformaties is over het algemeen een nieuwe, langere string. Door de regels herhaaldelijk toe te passen op de resulterende string kun je dus een simpel axioma systematisch uitbreiden. In hoofdstuk 3 zal de precieze werking van zo'n L-syteem worden uitgelegd.

Het resultaat van deze simulatie is dus een string, welke vaak lang is en moeilijk te interpreteren. Om deze simulaties nuttig te kunnen gebruiken is een goede visualisatie dus essentieel. Het laatste onderdeel van deze opdracht vraagt je daarom om de resultaten via een grafische bibliotheek te representeren op je scherm.

Fun fact: Omdat het Lindenmayer-systeem de groei van planten zo goed beschrijft en de uiteindelijke taal goed te visualiseren is, wordt dit systeem ook in de videogame-industrie gebruikt om bomen en planten te genereren. Een voorbeeld hiervan zie je in Figuur 1.

## 1.1 Leerdoelen

Aan het einde van deze opdracht kun je:

- omgaan met `.properties` bestanden,
- omgaan met een grafische bibliotheek,
- omgaan met stacks,
- een simpele simulatie implementeren,
- een beschrijving van functionaliteit omzetten tot een nette implementatie.

[1] `https://en.wikipedia.org/wiki/L-system`

Figuur 1: Een 3D boom gegenereerd met het Lindenmayer-systeem

# 2 Framework

Net als bij de voorgaande opdrachten, krijg je ook nu weer een framework aangeleverd. Dit framework handelt het opzetten van het grafische gedeelte van de opgave voor jou af, waardoor jij je kunt concentreren op de implementatie van de opdracht zelf.

Wat nu echter anders is, is dat dit ook het enige is wat je krijgt aangeleverd. Voor de rest van de opgave ben je vrij om het op jouw eigen manier te implementeren, zolang de functionaliteit zoals beschreven in de volgende hoofdstukken maar aanwezig is. Je kunt ons nu dus laten zien dat je begrijpt hoe je een beschrijving kunt omzetten tot een logisch opgebouwde, objectgeoriënteerde applicatie. De tekst in deze opgave zal je wel wat handvaten bieden om de functionaliteit te kunnen implementeren. Probeer deze functionaliteit zelf onder te verdelen in meerdere, logisch opgebouwde objecten.

## 2.1 Bestanden

Het framework bestaat uit één bestand en een mapje vol met invoer. De inhoud van het framework is als volgt:

`Opgave7.java` In dit bestand wordt de grafische gebruikersinterface aangemaakt.

`invoer/` De bestanden in dit mapje staat de invoer die je kunt gebruiken voor deze opgave. Je bent natuurlijk vrij om nog meer bestanden hieraan toe te voegen (en dan komen die misschien volgend jaar wel in het framework).

**Tip** Stop vooral **niet** je volledige implementatie in Opgave7.java.

# 3 Lindenmayer-systeem

In dit hoofdstuk wordt eerst kort uitgelegd hoe een Lindenmayer-systeem werkt. Zorg dat je dit concept goed begrijpt voor je verder gaat met de opgave.

Laten we beginnen met een voorbeeldje. Stel we nemen de volgende initiële waarden:

```
Alfabet: {A, B}
Axiom  : {A}
Regels : {{A: AB}, {B: A}}
```

Hier staat dus dat ons alfabet bestaat uit twee symbolen, namelijk `A` en `B`. We starten met de string `A`, en onze regels stellen ons in staat om een `A` te vervangen door `AB`, en een `B` door een `A`. Deze regels worden toegepast op de string waar je in die iteratie mee start, en niet op het tussenresultaat. Als voorbeeld, stel dat we dit systeem één keer uitvoeren, dan krijgen we dus

```
A -> AB
```

Belangrijk om hier op te merken is dat de tweede regel, `B: A`, hier dus niet wordt toegepast! De `AB` die door de eerste regel is geproduceerd, wordt namelijk in de resulterende string geplaatst.

Dit proces kun je verder herhalen op de resulterende strings. Hieronder staan de uitkomsten voor de volgende `n` iteraties ter illustratie.

```
(n = 0) | A
(n = 1) | A -> AB
(n = 2) | AB -> ABA
(n = 3) | ABA -> ABAAB
(n = 4) | ABAAB -> ABAABABA
(n = 5) | ABAABABA -> ABAABABAABAAB
```

Dit is een zogeheten Fibonacci woord[2]. Als je de lengte van elke string telt, krijg je dus ook de bekende Fibonacci reeks (minus de eerste 1, dit komt door het axioma): 1, 2, 3, 5, 8, 13….

Door het alfabet, het axioma en de regels uit te breiden is het mogelijk om complexe systemen op een simpele manier te modelleren. Echter, zoals je ziet neemt de lengte van de strings al vrij snel toe en is een visualisatie dus noodzakelijk.

[2] `https://en.wikipedia.org/wiki/Fibonacci_word`

# 4 Implementatie

## 4.1 Stap 1: Het uitlezen ven invoerbestanden

Voordat je kunt beginnen met het implementeren van het Lindenmayer-systeem, heb je eerst invoer nodig om te kunnen verwerken. Echter, dit keer ga je niet de Scanner gebruiken om een invoer bestand handmatig te parsen, maar ga je gebruik maken van een ingebouwde bibliotheek in Java die een deel van het werk voor je uit handen neemt.

In de `invoer/` map zitten een aantal `.properties` bestanden. Een voorbeeld is het bestand `dragoncurve.properties`, welke ook hieronder staat:

```
name=Dragon Curve

recursion.depth=15
axiom=FX
rules=X X+YF+,Y -FX-Y

turtle.pos.x=200
turtle.pos.y=300
turtle.angle=90.0
turtle.line.length=2
turtle.line.width=1
turtle.angle.modifier=90.0
```

De betekenis van deze eigenschap is als volgt:

- name: De naam van het figuur.
- recursion.depth: Hoeveel iteraties van het Lindenmayer-systeem je moet uitvoeren.
- axiom: De startwaarde voor het axioma.
- rules: De regels die toegepast moeten worden op het axioma.
  - rules kan meerdere regels bevatten. De regels zijn opgesplitst door komma's (,).
  - Een regel bevat twee elementen: Het symbool wat vervangen dient te worden en de symbolen waar het door vervangen moet worden. Deze zijn van elkaar gescheiden door spaties ( ).
- turtle.pos.x: De positie op de x-as waar de schildpad moet starten.
- turtle.pos.y: De positie op de y-as waar de schildpad moet starten.
- turtle.angle: De initiële kijkrichting van de schildpad.
- turtle.line.length: De grootte van een stap/lengte van een lijn.
- turtle.line.width: De breedte van een lijn.
- turtle.angle.modifier: De draai-hoek van de schildpad

De eigenschappen in een `.properties` bestand zijn opgeslagen als zogeheten `key-value paren`, waarbij elke eigenschap een eigen naam heeft. Denk hierbij ook bijvoorbeeld aan variabelen in jouw code. Het voordeel hiervan is dat de bestanden beter leesbaar zijn en, nog belangrijker, ze zijn adresseerbaar! Met behulp van de `Properties`[3] bibliotheek in Java kun je deze bestanden dus zeer gemakkelijk uitlezen. Hieronder staat een voorbeeld voor hoe je de `name` en `recursion.depth` kunt uitlezen.

[3] `https://docs.oracle.com/javase/10/docs/api/java/util/Properties.html`

```
Properties prop = new Properties();
InputStream input = new FileInputStream("dragoncurve.properties");

prop.load(input);
String naam = prop.getProperty("name");
input.close();
```

Gebruik het bovenstaande voorbeeld om alle andere eigenschappen uit te lezen. Je hoeft de eigenschappen voor nu nog niet te verwerken of splitsen, zorg er eerst voor dat je begrijpt hoe je een `.properties` bestand gebruikt.

**Tip** Controleer altijd of de invoer uit een properties bestand geldig is! Je kunt **niet** aannemen dat de invoer altijd helemaal klopt.

### 4.1.1 Stap 1.1: Verwerken van de invoer

Het zal je vast opgevallen zijn dat bijna alle eigenschappen binnen het `.properties` bestand een simpel datatype hebben, wat het verwerken makkelijk maakt. De enige uitzondering hierop zijn de `rules`. Elke regel bestaat uit twee delen, gescheiden door een spatie. Het eerste deel beschrijft de het symbool wat vervangen dient te worden, het tweede gedeelte beschrijft waarmee het eerdere symbool vervangen moet worden. Tevens kan `rules` meerdere regels bevatten, gescheiden door een komma.

Het opsplitsen van een string hebben jullie in eerdere opdrachten al gedaan, dus nu rest nog de vraag hoe je een individuele regel kunt opslaan. Met de kennis die jullie in eerdere opdrachten hebben opgedaan zou het misschien logisch klinken om een `Regel` object te maken welke de twee delen van de regel bevat, en dan de functionaliteit te implementeren om hier doorheen te zoeken wanneer je het Lindenmayer-systeem gaat toepassen om de uiteindelijke string te genereren. Maar, zou het niet veel handiger zijn als er binnen Java hier al iets voor bestaat?

In Java, maar ook in veel andere programmeertalen, bestaat het concept van `Maps`[4]. `Maps` (soms ook `Dictionaries`) zijn datastructuren, net zoals arrays en `ArrayLists`. Het verschil is echter dat de waardes in de tabel niet worden geïndexeerd aan de hand van oplopende integerwaardes, maar aan de hand van een `key`. Net als in de `.properties` bestanden sla je in `Maps` dus eigenschappen op als `key-value` paar.

Verwerk nu de `name`, `recursion.depth` en `axiom` eigenschappen. Lees de documentatie van de `Maps` klasse door en gebruik deze om vervolgens de regels die je hebt uitgelezen te verwerken en op te slaan. De eigenschappen beginnend met `turtle` mag je voor nu overslaan: hier komen we bij stap 3 op terug.

### 4.1.2 Stap 1.2: Genereren van de string

Je hebt nu alle eigenschappen die je nodig hebt om de uiteindelijke string te genereren:

- axiom: De begin-string
- rules: De regels die je toe moet passen op het axioma
- recursion.depth: Het aantal iteraties dat je moet uitvoeren

Genereer nu op **recursieve** wijze de uiteindelijke string aan de hand van de bovenstaande eigenschappen. Om dit te testen kun je de `recursion.depth` aanpassen naar een kleinere waarde, en dan met de hand controleren of het klopt. Ook zou je een eigen `.properties` bestand kunnen maken aan de hand van het voorbeeld in het vorige hoofdstuk. Zorg er in ieder geval voor dat je er zeker van bent dat het genereren goed gaat voor je doorgaat naar de volgende stap. De string die hier uit moet komen is de uiteindelijke string na `n` iteraties, waarbij `n` gedefinieerd is in het `properties` bestand.

[4] `https://docs.oracle.com/javase/10/docs/api/java/util/Map.html`

## 4.2 Stap 2: Visualiseren

Het Lindenmayer-systeem levert ons een string op met symbolen. Dit is leuk voor bijvoorbeeld de Fibonacci reeks, maar voor ingewikkeldere uitkomsten is het lastig om de (steeds groter wordende) strings te begrijpen. Om deze reden ga je nu de uitkomsten visualiseren.

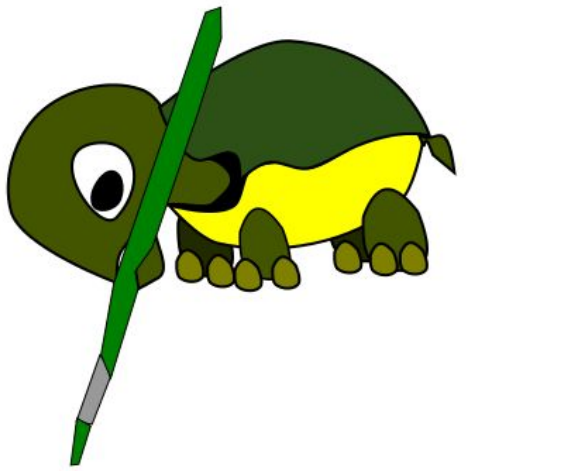
Bij het opstarten van het programma zul je hebben gezien dat er ook een venster verschijnt met daarin een lijn. Dit venster ga je gebruiken om de uitkomst in af te beelden. Bestudeer de code en het commentaar van `Opgave7.java`, en probeer zelf een aantal lijnen aan het venster toe te voegen. Het is belangrijk om te onthouden dat het punt `(X = 0; Y = 0)` **links-bovenin** zit. Deze zit dus niet in het midden, of zoals bij een traditioneel, wiskundig assenstelsel linksonder.

### 4.2.1 Stap 2.1: Turtle graphics

Nu je het tekenen op het scherm onder de knie hebt, is het tijd om te bedenken hoe je de Lindenmayer-string visueel kunt representeren. Een graphics-systeem dat zich hier uitermate goed voor leent zijn de zogeheten turtle graphics[5].

Turtle graphics werken als zijnde zogeheten (relatieve) cursor graphics. Waar je in de vorige stap handmatig coördinaten voor het begin- end eindpunt invoerde, en op deze manier lijn voor lijn een tekening kan maken, werken cursor graphics zoals de kwast in Microsoft Paint. Zodra jij de muisknop ingedrukt houdt, wordt er getekend daar waar je de cursor (muis) heen beweegt. De naam *turtle graphics* komt van de metafoor die wordt gebruikt om dit systeem uit te leggen: stel je een schildpadje voor dat op je scherm staat. Op het schild van deze schildpad zit een stift vastgemaakt, en de punt van deze stift sleept de schildpad achter zich aan. Door nu de schildpad te laten manoeuvreren over het scherm, ontstaan er dus tekeningen op het canvas.

Maak nu allereerst een schildpad aan met behulp van de eigenschappen in het `.properties` bestand. De betekenis van de `turtle.pos.x`, `turtle.pos.y` en `turtle.angle` spreken voor zich. De `turtle.line.length` en `turtle.line.width` geven aan hoe lang en breed een lijn (dus stap) van de schildpad zijn. De `turtle.angle.modifier` is hoeveel graden de schildpad moet draaien bij een bocht.




[5] `https://en.wikipedia.org/wiki/Turtle_graphics`

### 4.2.2 Stap 2.2: De Lindenmayer string parsen

Je hebt nu een schildpad, maar deze kan nog niet zo heel veel. De mechanismen om deze aan te kunnen sturen missen namelijk nog.

De reden dat turtle graphics zo goed werken met het Lindenmayer-systeem komt vanwege de uiteindelijke string (taal) die Lindenmayer genereert, welke bestaat uit een collectie van symbolen. Door elk van deze symbolen een betekenis te geven voor onze schildpad kun je de schildpad de taal laten lezen, om vervolgens de instructies uit te voeren. Hieronder staat de vocabulaire van de schildpad gedefinieerd:

- F: Zet een stap (teken een lijn)
- G: Zet een stap (teken een lijn)
- −: Draai naar links
- +: Draai naar rechts

De F en G kunnen door elkaar heen gebruikt worden. Ze hebben voor de schildpad dezelfde betekenis, maar het onderscheid is belangrijk voor het Lindenmayer-systeem. Overige symbolen hebben geen betekenis: hier hoeft dus niets mee te gebeuren.

Onthoud bij het zetten van een stap dat dit niet alleen maar horizontaal of verticaal kan zijn, maar ook onder een andere hoek. Omdat we begrijpen dat de bijbehorende wiskunde mogelijk wat is weggezakt, staat hieronder de formule waarmee je de afgelegde afstand bij een stap kunt berekenen.

```
x = x + <stapgrootte> * cos(<hoek in radialen>)
y = y + <stapgrootte> * sin(<hoek in radialen>)
```

In het framework zijn wat aannames gemaakt over hoe de methode die je aanroept gaat werken, deels om je ook naar een goede oplossing te sturen. Allereerst is er een variabele `startProcessing`, welke in de `main` van `Opgave7.java` op `true` wordt gezet. Dit dient vanaf nu pas te gebeuren zodra de uiteindelijke taal door het Lindenmayer-systeem is gegenereerd. Mocht dit niet goed gebeuren, dan kan het zo zijn dat er helemaal niks getekend wordt, omdat de `paint` methode in `Opgave7.java` soms aangeroepen wordt nog voordat de generatie van de taal klaar is. Wat er vervolgens dient te gebeuren is dat de `paint` methode een methode aanroept die een enkel karakter uit de taal verwerkt. Het resultaat uit deze methode is of een `Line2D.Double` object welke getekend kan worden, of `null`. Zodra de hele string verwerkt is, dient `startProcessing` weer op false te worden gezet. (Kun je een nette manier verzinnen om het einde te detecteren?)

Als je het bovenstaande hebt geïmplementeerd is het mogelijk om de applicatie uit te voeren met `koch.properties`, `sierpinski.properties` en `dragoncurve.properties`, waarna je (als je alles goed hebt gedaan) afbeeldingen te zien krijgt zoals in de voorbeelduitvoer hieronder.

## 4.3 Stap 3: De stack

Op dit moment kun je al figuren tekenen. Het nadeel van deze figuren is echter dat ze altijd moeten bestaan uit één lange lijn. Bomen en andere natuurlijke verschijnselen waarin vertakkingen voorkomen tekenen wordt hiermee dus heel lastig. Om deze reden moet je de schildpad gaan uitbreiden met wat extra functionaliteit.

Allereerst even een idee van de functionaliteit. Wat je hier dient te bereiken is dat de schildpad ook vertakkingen, zoals in bomen en planten, kan tekenen. Om even bij de metafoor van de schildpad te blijven, stel dat je een vertakking in de vorm van de letter 'Y' hebt. Wat je dan wilt is dat de schildpad op het punt van de vertakking, bijvoorbeeld, eerst naar links draait en de linker tak tekent. Vervolgens pak je zelf de schildpad op, en zet je deze weer neer op het punt van de vertakking. De schildpad draait vervolgens naar rechts en tekent de rechter kant van de vertakking.

Om dit op een effectieve manier te bereiken, dien je een zogeheten stack te maken. Een stack is een simpele datastructuur om een reeks aan waardes op te slaan. Het idee achter een stack is vergelijkbaar

met een stapel dozen. Als je iets `push`t naar de stack, dan zet je een doos **bovenop** de stapel. Als je vervolgens iets `pop`t van de stack, dan pak je de bovenste doos van de stapel. Het element dat je pakt is dus altijd het laatste element dat is toegevoegd.

Implementeer nu eerst een stack om de staat van de schildpad naartoe te pushen of vanaf te poppen. Bedenk hierbij goed welke waardes je van de schildpad dient te onthouden om de bovenstaande functionaliteit te realiseren, en hoe je deze gaat opslaan.

Vervolgens breiden we de vocabulaire van de schildpad als volgt uit:

- [: Push de huidige staat van de schildpad naar de stack
- ]: Pop de laatst bekende staat van de schildpad van de stack

Als je de stack correct hebt geïmplementeerd, is je applicatie nu in staat om `tree.properties` en `fractalplant.properties` te tekenen. Vergelijk ook de uitvoer van deze bestanden met de voorbeelden hieronder.

### 4.4 Extra's

Gefeliciteerd! Als je hier bent gekomen, dan is het je gelukt om een correct werkend Lindenmayersysteem te implementeren, en deze te visualiseren met behulp van turtle graphics! Mocht je nog op zoek zijn naar wat extra uitdaging (of punten), dan kun je nog kijken naar de volgende aspecten. Let wel op: begin hier pas aan als de rest van de functionaliteit is geïmplementeerd. Anders krijg je hier geen extra punten voor:

- Maak een extra optie voor de applicatie die ervoor zorgt dat de visualisatie in slechts een enkele stap wordt getekend. Deze optie dient configureerbaar te zijn via bijvoorbeeld de CLI of de `.properties` bestanden.
- Voeg functionaliteit toe om de lijnen van dikte en lengte te laten verschillen. Denk hierbij bijvoorbeeld aan een boom waarbij de takken met elke stap korter en dunner worden. Voeg ook een (aangepaste) `.properties` bestand toe om de functionaliteit aan te tonen.
- Voeg functionaliteit toe om met verschillende kleuren lijnen te werken. Denk bijvoorbeeld aan een boom met een bruine bast en groene bladeren. Voeg ook een (aangepaste) `.properties` bestand toe om de functionaliteit aan te tonen.

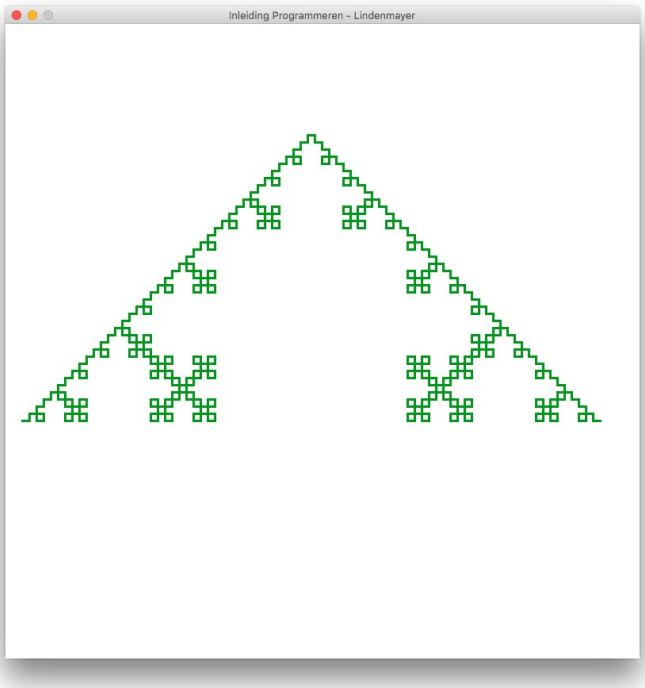
## 5 Inleveren

Stop jouw `.java` en `.properties` bestanden in een archief. De geaccepteerde bestandsformaten zijn `.zip`, `.tar` en `.tar.gz`. Lever dit bestand in.

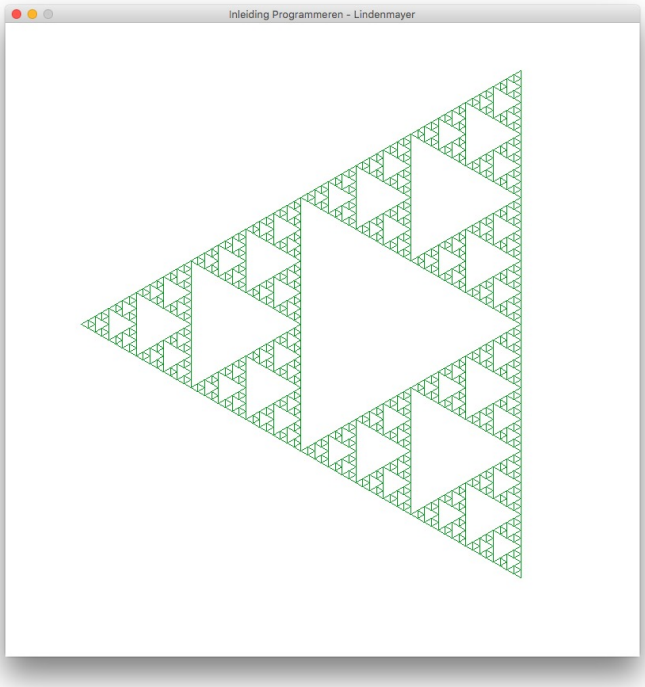
**Let op: lever geen andere bestandsformaten in! Dit zorgt ervoor dat we jou geen volledige feedback kunnen geven!**

## 6 Voorbeelduitvoer

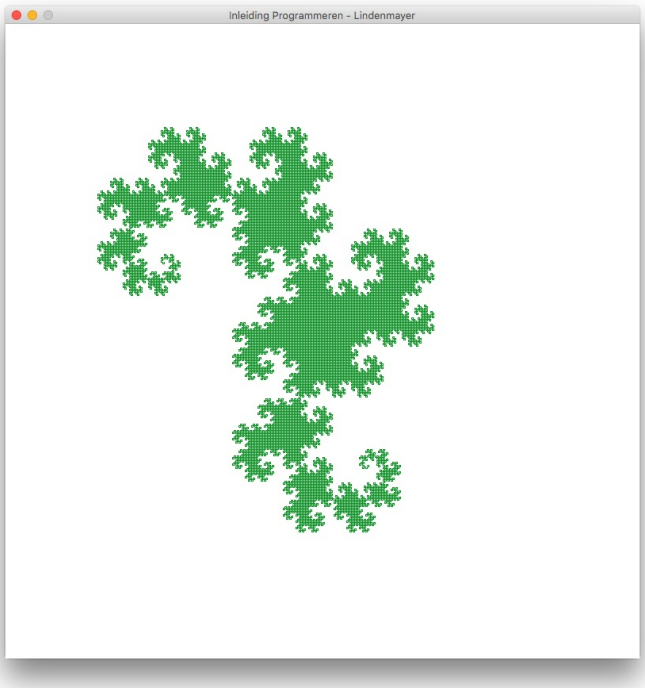
Hieronder staat de uitvoer voor de vijf verschillende .properties bestanden in het framework afgebeeld. Controleer goed of jouw uitvoer overeenkomt.



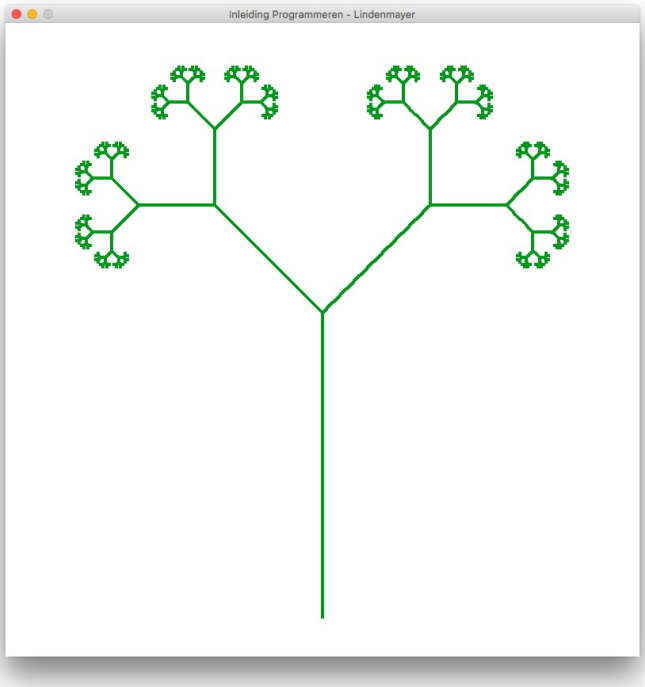

Figuur 2: koch.properties



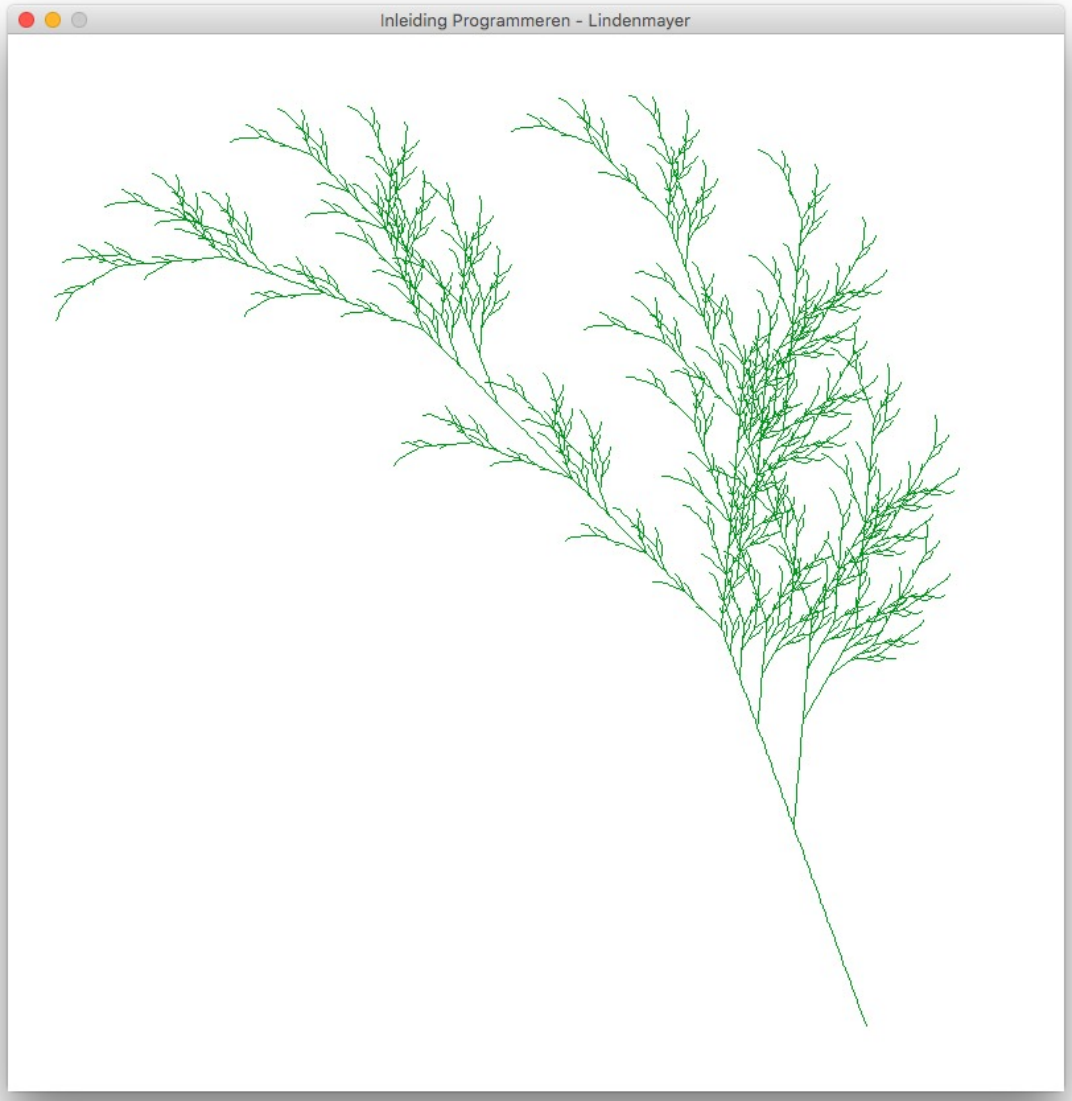

Figuur 3: sierpinski.properties




Figuur 4: dragoncurve.properties




Figuur 5: tree.properties




Figuur 6: fractalplant.properties