Hebben constructors overerving?

Ja

Nee

Dat ligt er aan

```java
class P {
    String m() {
        return "P!";
    }
}

class C extends P {
    String m() {
        return "C!";
    }
}

P p = new C();
System.out.println(p.m());
```

P!

C!

compileert niet

Wat is het resultaat?

```
class Cheese {

    public String name;
    public int age;

    Cheese() {...}

    Cheese(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void set_name(String Name) {
        this.name = Name;
    }

    String get_name() {
        return name;
    }
}
```

Wat is er allemaal mis met deze code? Wat zijn de ergste fouten?

```java
public class Test
{
    static class A {
        static void m() {
            System.out.println("X");
        }
    }

    static class B extends A {
    }

    public static void main(String[] args)
    {
        B.m();
    }
}
```

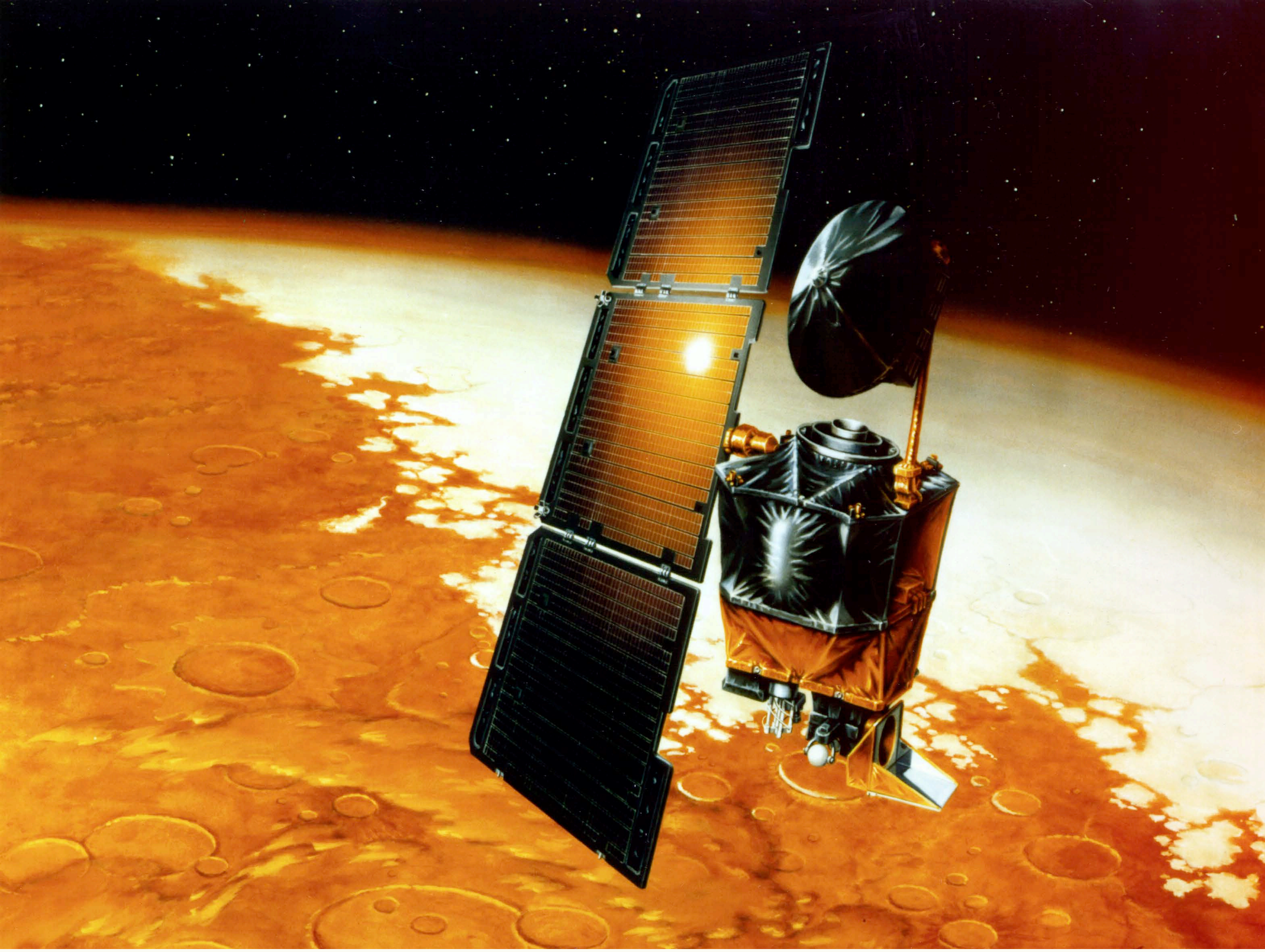
“X”

compileert niet

# Hoe schrijf ik VEILIGE code?

java@peterbloem.nl

Dit is de explosie van de eerste Ariane 5 raket, in 1996, ter waarde van 370 miljoen dollar. De raket gebruikte  deels dezelfde software als zijn voorganger, maar had een grotere acceleratie. Hiervoor werden double waarden gebruikt (64 bits floating point nummers), en in de oude versie werden 16 bits integers gebruikt. Bij de conversie van één van die getallen werd niet gecontroleerd of de double waarde wel paste in een 16 bits integer.



Dit is de Mars Climate Orbiter. Een mars satteliet van NASA die in 1998 het contact met de aarde verloor. Eén van de systemen produceerde waarden in engelse eenheden (inches, feet etc) terwijl een ander system metrische waarden verwachtte (kilometers, etc). Wederom, een prijs-kaartje van zo'n 320 miljoen dollar.

Dit is de Therac-25. Een bestralingsmachine voor radiotherapie. De machine kon bestralen met een electronenstraal op lage intensiteit, en met Rontgenstraling, waarbij de intensiteit van het electronenkanon verhoogd werd naar een levensgevaarlijk nivea, maar er een schild in de loop van de electronen werd geplaatst dat de electronen naar Rontgenstraling converteerde.

De hardware had (in tegenstelling tot eerdere versies) geen safeguards die zorgden dat het electronenkannon niet kon werken zonder schild (de maker had besloten om volledig op de software te vertrouwen). Een gebrek aan synchronisatie zorgde er voor dat een vingervlugge bediener het electronen kannon aan kon zetten voordat het schild op zijn plek stond. Daarnaast incrementeerde de software een integer om een state in het systeem aan te geven. Als deze waarde overflowde kon het electronenkanon gestart worden zonder schild.

Verschillende mensen kregen honderden malen de bedoelde dosis straling en ten minste drie patienten overleden als een direct gevolg.

Dit is Stanislav Petrov, oud-lieutenant van de Soviet luchtmacht. In de herfst van 1983, waren de relaties tussen de VS en de Soviet-Unie uiterst gespannen. De Soviet-Unie had net een passagiersvliegtuig neergehaald dat haar luchtruim had gekruisd en een van de overledenen was een Amerikaans congresslid. Op 26 September kreeg Petrov meldingen binnen dat de Amerikanen twee intercontinentale nuceaire raketten hadden afgevuurd.

De doctrine van de Soviet-Unie leunde destijds erg op het principe dat een tegenaanval zonder aarzeling uitgevoerd moest worden. Petrov redeneerde dat een aanval van de Amerikanen met honderden raketten tegelijk zou komen en niet met een hand vol. Hij besloot de waarschuwing niet door te geven aan zijn bevelhebber.

Uiteindelijk bleek de waarschuwing een computerfout te zijn. De details zijn niet bekend, maar het waarschuwingssysteem bleek vol bugs te zitten.

# Wat kost een bug?

| | |
|---|---|
| ontwikkeling | 10€ |
| integratie | 100€ |
| acceptatie | 1 000 € |
| productie | vanaf 10 000 € |

Een standaard software-ontwikkelproces is in vier fasen opgedeeld, de *OTAP* straat: ontwikkeling (de programmeur achter zijn computer), integratie (of testing, een server bij de ontwikkelaar waar de software van alle programmeurs bij elkaar wordt gebracht), acceptatie (een server bij de klant, waar de klant kan testen) en productie (het uiteindelijke systeem).

Als de ontwikkelaar een bug in zijn code vindt is het doorgaans in een kwartiertje opgelost. Als het tijdens integratie fout gaat moeten alle programmeurs meedenken. Als het tijdens acceptatie fout gaat moet ook de klant er tijd in stoppen, en moet de hele straat weer doorlopen worden. En, zoals we gezien hebben, een productie-bug kan desastreuze gevolgen hebben.

# Veilige code: 3 principes

1. Vroege errors zijn beter dan late errors
2. Beperk toegang tot informatie zoveel mogelijk
3. Maak je code modulair

Dit zijn de drie belangrijkste principes om je code veilig te krijgen.

In de eerste plaats is het belangrijk dat wanneer er een fout gemaakt wordt je daar zo vroeg mogelijk achter komt. Zoals we hebben gezien nemen de kosten van een bug exponentieel toe met hoe laat hij gevonden wordt.

Ten tweede is het belangrijk om toegang streng te beperken. Hoe een object intern zijn informatie opslaat en welke informatie het blootstelt kunnen enorm verschillen. Zoals we met de Date klasse hebben gezien kan het blooststellen van de verkeerde informatie ernstige gevolgen hebben.

Tot slot is modulaire code ook hier belangrijk. Zorg dat ieder stukje code (een object, een methode, etc.) een enkel en simpel doel heeft. Op die manier kun je de bouwblokken eerst grondig afzonderlijk testen voordat je er mee verder gaat. Als je toch een bug in je productiecode ontdekt is het daarmee ook veel simpeler om op te sporen waar het probleem zit.

```
class Test {
    public static void main(String[] args)
    {
        int mySpecialInteger = 0;
        int mySecondSpecialInteger = 0;
        int myResultInteger = 0;

        for (int myIteratingInteger : Arrays.asList(0, 1, 3))
        {
            mySpecialInteger +=
                            myIteratingInteger + myResultInteger;
            mySecondSpecialInteger += myIteratingInteger *
                                      mySpecialInteger;

            myResultinteger = myIteratingInteger *
                mySpecialInteger % mySecondSpecialInteger;
        }

        System.out.println(myResultInteger);
    }
}
```

Dit is een ingewikkeld stuk code. Wat gebeurt er als we deze code proberen te draaien?

```
class Test {
    public static void main(String[] args)
    {
        int mySpecialInteger = 0;
        int mySecondSpecialInteger = 0;
        int myResultInteger = 0;

        for (int myIteratingInteger : Arrays.asList(0, 1, 3))
        {
            mySpecialInteger +=
                            myIteratingInteger + myResultInteger;
            mySecondSpecialInteger += myResultInteger *
                                      mySpecialInteger;

            myResultinteger = myIteratingInteger *
                mySpecialInteger % mySecondSpecialInteger;
        }

        System.out.println(myResultInteger);
    }
}
```

# In Python

```
my_special_integer = 0
my_second_special_integer = 0
my_result_integer = 0

for my_iterating_integer in [0, 1, 5]:
    my_special_integer += my_iterating_integer + \
                                              my_result_integer
    my_second_special_integer += my_result_integer * \
                                              my_special_integer
    my_resulting_integer = my_iterating_integer * \
        my_special_integer % my_second_special_integer

print my_result_integer
```

In andere talen (zoals python) wordt het je makkelijker gemaakt en hoef je variabelen niet van te voren te declareren. Dit betekent dat je deze fout pas tegenkomt als je het programma draait.

Dit betekent niet dat Python een slechtere taal is dan Java (of andersom), maar het is belangrijk om te weten waarom je in Java vaak wat extra moet typen. Uiteindelijk is het uittypen van je programma niet waar de meeste tijd in gaat zitten, dus extra typewerk voor extra veiligheid is geen slechte deal.

Overigens wordt het in Python afgeraden on zulke lange variabele namen te gebruiken, onder andere omdat de kans dan groter is dat je een typfout maakt.

# Early errors: voorproefje van Exceptions

```
class Person
{
    private age;

    void setAge(int age)
    {
        if (age < 0)
            throw new IllegalArgumentException(
                        "Age ("+age+") can't be negative.");
        this.age = age;
    }
    ...
}
```

## Alle objecten die inheriten van `RuntimeException` kun je op deze manier gebruiken

Java probeert zoveel mogelijk bugs als compile time errors te vangen, maar dat lukt niet altijd. Voor runtime bugs geldt ook dat je ze altijd zo snel mogelijk wilt signaleren. Daarvoor gebruiken we exceptions. Daar gaan we later uitgebreid op in, maar hier vast een voorproefje. Met het throw keyword “gooi” je een Exception object. Dit komt er in principe op neer dat het programma gestaakt wordt met de opgegeven boodschap.

In dit geval kan het voorkomen dat iemand een persoon een nagatieve leeftijd geeft. Als we die input niet controleren, krijgen we pas een error als de leeftijd ergens gebruikt wordt. En misschien niet eens een runtime error, misschien wel onverwacht gedrag. Daarom controleren we de input en “gooien” we een Exception. Het programma stopt met een foutmelding zodra er een negatieve leeftijd gebruikt wordt.

Je kunt exceptions ook zelf afvangen en afhandelen, zodat je programma niet hoeft te eindigen. Daar komen we later op terug.

# compile-time errors zijn altijd beter dan runtime errors

## runtime errors zijn (bijna) altijd beter dan onverwacht gedrag

Dit geeft een belangrijk onderscheid aan. Veel van de dingen die Java wat tijdrovender maken om te programmeren, zijn er om te zorgen dat bugs **compile-time** worden in plaats van run-time. Compile time bugs hebben veen voordelen:

- Je ziet ze altijd in de ontwikkelfase. Runtime bugs kunnen op elk punt in de OTAP straat voorkomen.
- De compiler kan aangeven waar de fout zit, en wat je er aan kunt doen. Een runtime bug kan op een heel ander punt ontstaan dan waar het uiteindelijk fout gaat

Kortom, als je een programmeerfout maakt wilt je dat zo snel mogelijk merken en de plek in de code waar je het merkt moet zo dicht mogelijk bij de foute code liggen.

# 2) Beperk de toegang

```
public class Date {
    private int day;
    private int month;
    private int day;

    public int getDay(){...}
    public int getMonth(){...}
    public int getYear(){...}

    public void setDay(int d){...}
    public void setMonth(int m){...}
    public void setYear(int y){...}
}
```

Dit principe zijn we al eerder tegengekomen, maar het is het waard om te herhalen. Het is belangrijk om de toegang tot je ruwe data zoveel mogelijk te bepreken. Dit heet information hiding, of encapsulation.

Hier nog een keer de Date klasse van vorige week. De velden zijn private, maar met deze setters kan een gebruiker nog steeds een illegale datum aanmaken. Iedere setter moet dus zorgvuldig de invoer controleren. En wat als ik eerst de dag op 31 januari zet, en dan de maand op februari?

In veel gevallen is het beter om de setters helemaal weg te laten. Als mensen een datum aan willen passen, dan krijgen ze een nieuw Datum object. Dit heet een immutable klasse.

# fields altijd `private`

## (of `protected`)

Ook als je klassen wel mutable zijn, fields maak je altijd private. Als je Java schrijft, maak je nooit je fields public.

Protected mag, maar eigenlijk is het in dat geval ook beter om ze private te maken en er protected getters en setters voor te schrijven.

```
class Person {
    private Date birthday;

    ...

    Date getBirthday() {
        return birthday;
    }

    void setBirthday(Date b) {
        this.birthday = b;
    }
}

class Date {
    ...
    public void decrement()
    {...}
}
```

```
// check if birthday is tomorrow
Cake cake;

while (true) {
    if (john.getBirthday().decrement()
        .equals(TODAY))
        cake = buyCake();

    if (john.getBirthday()
        .equals(TODAY))
        cake.giveTo(john);
}
```

wat gaat hier fout?

# Immutable klassen

1. Ongevaarlijk om met onvertrouwde processen te delen
2. Gedeelde internals zijn veilig
3. Interning voor efficiëntie
4. Altijd thread-safe
5. nadeel: apart object voor iedere waarde

oplossing, mutable companion: bijv. **`String`** & **`Stringbuilder`**

Immutable klassen hebben veel voordelen. Je kunt het rustig delen met een ander proces zonder bang te zijn dat het proces het object aanpast.

Je kunt zelfs de interne waarden van een object delen . Ik kan verschillende Date objecten maken die verwijzen naar de interne long van een basis date object, omdat ik zeker weet dat de verwijzing veilig blijft.

Immutable objecten kun je met een statische functie aanmaken ipv. een constructor en zorgen dat er per waarde maar één object aangemaakt wordt. Java doet dit automatisch met Strings. Dit heet interning.

Als er verschillende processen tegelijk een object aanpassen, kan het proces in een illegale state terechtkomen. Met immutable klassen heb je dit probleem niet.

Het enige nadeel van een immutable klasse is dat je per waarde een object nodig hebt. Als je bijvoorbeeld een immutable object hebt met een heel groot getal van een miljoen bits, dat je met één op wilt hogen, moet je het hele object kopiëren. In dat geval wordt vaak een mutable versie van het object gemaakt, waar je de immutable  variant uit kunt halen.

# Immutable classes: 5 principes

klassen final

```
public final Date {
    private final long dSince1Jan1970;

    public Date(int d, int m, int y)
    {...}

    public int getDay(){...}
    public int getMonth(){...}
    public int getMOnth(){...}
}
```

fields final & private

geen setters en andere mutators

```
public final Planning {

    private final Date[4] milestones;
    private final Person manager;

    public Date(
        Date m1, Date m2,
        Date m3, Date m4,
        Person manager)
    {...}

    public Date[] getPlanning()
    { return milestones.clone(); }

    public Person manager()
        { new Person(manager); }
}
```

interne mutables defen-sief kopiëren

Om een class immutable te maken moet je deze **vijf** dingen doen

1. Alle fields final, hiermee kan de waarde niet veranderen nadat hij voor de eerste keer gezet is.
2. Alle fields private, zodat niemand er bij kan.
3. Geen methoden die de fields aanpassen (mutators of setters)
4. Maak de class zelf ook final. Hiermee mogen er geen subklassen gemaakt worden.
5. Geen toegang tot mutable fields. In dit geval zijn de fields milestones en manager mutable. Dat deze fields final zijn, betekent alleen dat we de referentie niet kunnen veranderen naar een ander object of array. Het object en het array kunnen we nog wel aanpassen. Als we dat dus aan de gebruiker geven, kunnen ze de state van het object aanpassen. Om dit te voorkomen moeten we defensief kopiëren

# Klassen ontwerpen

- Begin immutable
- Als immutable niet lukt, minimaliseer dan de mutability
- Het is makkelijker om later extra toegang te geven dan om later toegang weg te halen.
- Hetzelfde geldt voor features:

*when in doubt, leave it out*

# Objecten kopiëren: Object.clone()

1. Atypisch gebruik van interfaces
2. Cloneable interface past gedrag van superclass methode aan
3. Objecten worden aangemaakt zonder dat een constructor aangeroepen is
4. Zwak contract dat moeilijk correct te implementeren is
5. Incompatibel met final fields

In Java moet je expliciet aangeven dat een object gekopieerd mag worden. Dit is erg prettig voor de veiligheid van je code. Zo kun je bijvoorbeeld *singletons* maken: klassen waar altijd maar één object van bestaat.

Als je een object kopieerbaar wilt maken kan dat op verschillende manieren. De oorspronkelijnke manier was met Object.clone(). Als je klasse de interface Cloneable implementeert, dan maakt de clone methode automatisch een kopie van je object (door alle velden te kopiëren naar een nieuwe instantie). Je kunt de clone methode ook overriden.

Deze slide geeft vijf redenen waarom dit principe geen geslaagde uitvinding is en over het algemeen niet meer gebruikt wordt. Er zijn maar heel weinig situaties waarin je Cloneable wilt gebruiken. Als je toch in zo'n situatie terecht komt, zorg dan dat je zorgvuldig bent. Er zijn specifieke regels voor hoe je te werk moet gaan. We gaan er hier niet oop in, maar je kunt ze bijvoorbeeld vinden in *Effective Java*, item 11.

# Beter: de copy constructor

```
public class Flub {
    private final List<String> glob;
    private int blorg;

    public Flub(Flub other)
    {
        this.glob =
            new ArrayList<String>(
                other.glob());
        this.blorg = other.blorg();
    }

    public List<String> glob() {...}
    public int blorg() {...}
    public void change() {
        blorg ++; }
}
```

```
/**
 * Voor interfaces en subklassen
 */

List<String> list =
            new LinkedList<String>();
list.add("a");
list.add("b");
list.add("c");

// converteer naar ArrayList
list = new ArrayList<String>(list);

// converteer naar Set
Set<String> set =
        new HashSet<String>(list);
```

Een beter principe is om een copy constructor te maken. Hiermee gebruik je geen rare principes die buiten de standaard features van Java vallen, je weet zeker dat alle objecten die in omloop zijn door één van je constructors gemaakt zijn, en je kunt rustig final fields gebruiken. Je zit ook niet vast aan een contract, dus je kunt zelf bepalen of je copy constructor een deep copy or een shallow copy maakt.

Merk op dat Flub een mutable klasse is. Voor immutable klassen heeft het weinig zin om een copy-constructor te maken, omdat je altijd zonder problemen het oorspronkelijke object kan blijven gebruiken.

Het mooie van een copy constructor is dat de input een subklasse van Flub mag zijn, maar dat het resultaat altijd een Flub instantie is. Dit betekent dat de gebruiker de copy constructor kan aanroepen om objecten te converteren naar een Flub object. Alle collecties (List, Set) hebben bijvoorbeeld constructors om een arbitraire Collection te converteren naar het huidige object.

# 3) Modulariteit: Loose Coupling

```java
// te veel informatie
public static void sort(ArrayList list) {...}

// net genoeg informatie
public static void sort(List list) {...}

// te weinig informatie
public static void sort(Collection list) {...}
```

Loose coupling is het idee dat modules zo min mogelijk over elkaar moeten weten. Alles wat de gebruiker van een klasse over de klasse weet, moet in het achterhoofd gehouden worden als we de klasse aan willen passen. Hoe minder de gebruikende klasse weet hoe beter.

In dit voorbeeld zien we een sort methode. Als we die voor een ArrayList schrijven, hebben we twee problemen. Ten eerste kunnen we andere soorten lijsten niet sorteren zonder een nieuwe methode te schrijven en ten tweede moeten we deze methode aanpassen wanneer we de ArrayList klasse aanpassen. Het kan zijn dat de de methode blijft werken, maar we moeten het altijd even controleren.

Als we een de methode schrijven voor een List, hebben we deze nadelen niet. We hebben net genoeg informatie en toegang om een sorteermethode te schrijven. We kunnen alle Lists sorteren (ook Lists die nog niet bestaan), en we kunnen rustig ArrayList aanpassen, zolang hij de List interface maar blijft implementeren.

# Modulariteit: interfaces

```
public class PhoneNumber implements
    // Met andere objecten vergelijken
    Comparable,
    // Wegschrijven naar de harde schijf
    Serializable,
    // Ik kan het nummer bellen
    Callable,
    // Heeft een geografische locatie
    HasLocation
{...}

public class House implements  HasLocation {...}

public void plotOnMap(HasLocation[] locations)
{...}
```


Interfaces zijn ideaal om Loose Coupling voor elkaar te krijgen. Met een interface kun je, veel meer dan met inheritance een klein stukje van de de functionaliteit van een klasse isoleren.

Hier zie je wat voorbeelden van hoe interfaces gebruikt worden om een deel van de functionaliteit van een klasse te beschrijven. Ik kan nu een methode schrijven om dingen die een locatie hebben op een kaart te plotten, zonder de details te hoeven weten.

Net als bij de sort methode weet mijn plotOnMap methode niets meer dan nodig is. Ik kan rustig de methode aanpassen die een telefoonnummer belt zonder de plotOnMap methode in mijn achterhoofd te hoeven houden, want ik heb een garantie dat die alleen de methodes in PhoneNumber aanroept die door HasLocation voorgeschreven worden.

# Vragen?

Na de pauze gaan we het hebben over Generics en Exceptions

# Generics

## Voor Java  1.5: Alle collections bevatten Objecten

```
List stamps = ...;

stamps.add(new Coin());
stamps.get(0).paperType()
// compiler fout
```

```
List stuff = ...;

stuff.add("x");

String s = (String) stuff.get(0)
s.toLowerCase();
```

```
List stuff = ...;

stuff.add("x");

if (stuff.get(0) instanceof String)
{
    String s = (String) stuff.get(0)
    s.toLowerCase();
} else
    ...
```

Dit is hoe het werkte voor Java 1.5. Collections bevatten objecten. Als je iets uit een collectie haalt is de type informatie van de objectreferentie weg. Als je bijvoorbeeld linksonder de String methodes aan wilt roepen, moet je de Object referentie casten naar een String referentie.

Als je dit verkeerd doet en per ongeluk een object cast naar een String dat geen String is, dan krijg je een ClassCastException en stopt je programma. Als je het niet zeker weet, kun je instanceof gebruiken. Hiermee controleer je of je object een intsantie is van een gegeven klasse. Als je object een subklasse is van de gegeven klasse, geeft instanceof oof true terug (x instanceof Object is bijv. altijd waar).

Het is vervelend om steeds te moeten casten, maar er is nog een veel groter probleem met deze aanpak. Als je een fout maakt, merk je dat pas tijdens runtime. Zoals we gezien hebben willen we het liefst meteen weten als we iets fout doen, en het merken waar we het fout doen. Als we een verkeerd object in een lijst stoppen merken we dat tijdens runtime, en pas wanneer we het uitlezen, ver van de plek waar de fout daadwerkelijk gemaakt is.

# Generics

```
List<String> stuff = new ArrayList<String>();

stuff.add("x");

// geen probleem
stuff.get(0).toLowerCase();
String s = stuff.get(0);

// dit mag niet meer
stuff.add(new Double(1.0));
```

Met Generics ziet het er zo uit. Je geeft tussen de spekhaken <> aan welk type objecten in de Lijst geplaatst mogen worden. Als je het object uit de lijst haalt kun je dus meteen de methoden van dat type object aanroepen. De objectreferentie die je terug krijgt is van het type dat je eerde gespecificeerd hebt.

Als je iets fout doet merk je dat op het moment dat je het fout doet (tijdens het schrijven van de code), op de plek waar het fout gaat.

# Generics

```
class ArrayList<E>
{
    private E[] internal;

    public ArrayList(){...}

    public E get(int i){...}

    public add(E element){...}
}

public static <E> List<E> convertToArray(List<E> list)
{
    ....
}
```

Dit is hoe generics er van binnen uitzien. Je geeft achter het object een speciale variabele aan (E) die staat voor een Type (dwz. een klasse). Hiermee maak je in feite oneindig veel verschillende varianten van je klasse. Voor de klasse List<Blorg> wordt overal waar E staat “Blorg” ingevuld. Overal waar je klasse een “E” teruggeeft wordt dit automatisch gecast naar een “Blorg”.

De echte ArrayList klasse is wat complexer dan dit, maar het principe blijft hetzelfde.

Soms wil je voor een (statische) methode een generic type declareren. Dat kan ook, zoals hier getoond wordt.

Overigens werkt de echte ArrayList niet met een Generic array maar met een Object array. We gebruiken hier `E[]` om te laten zien dat het kan, de redenen dat ArryList het niet doet zijn vrij technisch (je kunt een generic array niet zomaar aanmaken).

# Voorbeeld: afstanden

```java
public interface Distance<E> {
    /**
     * Returns the distance between the given objects
     */
    public double distance(E first, E second);
}

public class EudclideanDistance implements Distance<Point> {
    public double distance(Point f, Point s) {
        double dx = f.x() - s.x(),
        double dy = f.y() - s.y();

        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Generics zijn niet alleen bedoeld om aan te geven wat er “in” een object zit. Voor allerlei relaties tussen objecten kun je Generics gebruiken.

In dit voorbeeld definieren we een afstandsmaat. Voor de Euclidische afstand geldt dat die alleen is gedefinieerd voor Punten, dus dat kunnen we in de klasse vastleggen.

# Voorbeeld: Afstanden

```
public class StringDistance<E> implements Distance<E>
{
    public double distance(E f, E s)
    {...}
}
```

Als we een String representatie hebben kunnen we een string afstandsmaat implementeren: bijvoorbeeld de edit distance. Dit kan op alle objecten, dus hoeven we het type van E niet vast te leggen.

# Voorbeeld: Afstanden

```
public static <E> double
                    meanDistance(List<E> things, Distance<E> dist)
{
    double sum = 0.0;
    for (E first : things)
        for (E second : things)
            sum += dist.distance(first, second);

    return sum / (things.size() * things.size());
}
```

Door bij een methode een type variabele aan te geven, kunnen we duidelijk maken dat de typen van twee objecten overeen moeten komen.

# Voorbeeld: Afstanden

```
List<Point> points = ...;
List<Blorgs> blorgs = ...;

Distance<Point> eucDist = new EuclideanDistance();
Distance<Blorg> strDist = new StringDistance<Blorg>();

// dit kan niet
Distance<Blorg> eucDist2 = new EuclideanDistance();
// dit wel
Distance<Point> strDist2 = new StringDistance<Point>();

double m;
m = meanDistance(points, eucDist);
m = meanDistance(blorgs, strDist);
// dit kan niet
m = meanDistance(blorgs, eucDist);
```

We kunnen nu specifieke versies van onze generieke aftsandsmaat aanmaken.

# Voorbeeld uit de SDK: Comparator

```
public interface Comparator<E> {
    public int compare(E first, E second);
}

public class AddressComparator implements Comparator<Address> {
    public int compare(Address first, Address second)
    {...}
}

public static <E> void sort(List<E> addresses, Comparator<E> c)
{...}

List<Address> addresses = ...;
sort(addresses, new AddressComparator());
```

Hier een voorbeeld uit de SDK. De Comparator. De comparator geeft een ordening over objecten aan. Voor twee objecten van het gegeven type geeft de comparator aan of a groter is dan b (negatief resultaat), b groter dan a (positief resultaat) of dat ze gelijk zijn (resultaat 0).

# Voorbeeld: Comparable

```
public interface Comparable<E>
{
    public int compareTo(E other);
}

public class Number implements Comparable<Number>
{
    public int compareTo(Number other){...}
}

public static
  <E extends <Comparable<? super E>> // dit bewaren we voor later
  void sort(List<E> addresses)
{
    ...
}
```

Sommige typen hebben een natuurlijke eigen sortering. In dat geval kun je de Comparable interface implementeren in het ibject zelf. Je kunt dan een sort methode maken zonder Comparator, omdat de objecten van zichzelf weten hoe ze zich met elkaar moeten vergelijken.

Je moet dan alleen in de type declaratie van sort(...) aangeven dat de objecten allemaal comparable moeten zijn. Dit moet met wildcards en bounds, een ingewikkelde feature. Hier gaan we het later over hebben.

```
static class A {
    void m(){ System.out.println(“A”);}}

static class B extends A {
    void m(){ System.out.println(“B”);}}

public static void main(String[] args) {
    List<A> first = new ArrayList<A>();
    List<A> second = new ArrayList<B>();
    List<B> third = new ArrayList<B>();

    first.add(new B()); second.add(new B()); third.add(new B());

    first.get(0).m();
    second.get(0).m();
    third.get(0).m();
}
```



A, A, B

B, B, B

Compileert niet

# Subtyping does not extend through Generics

```
public List<Number> list()
{
    return new ArrayList<Integer>();
}
```

`List<Integer>` is geen subklasse van `List<Number>`, hoewel `Integer` wel een subklasse is van `Number`.

# Exceptions: afhandelen

```java
public void method() {
    // do stuff...

    if (...) // something goes wrong
        throw new RuntimeException("AAH!");

    // other stuff...
}

public static void main(String[] args)
{
    method();
}
```

Exceptions hebben we al eerder gezien als nette manier om een programma te laten stoppen wanneer er iets fout gaat. Maar Exceptions kunnen een stuk meer. Hier zien we een voorbeeld . Als er in dit programma iets fout gaat in method(), dan stopt de uitvoer van het programma.

Maar stel nu dat we dat niet willen? Stel nu dat we zelf willen kijken of we toch nog door kunnen gaan met de uitvoer van het programma?

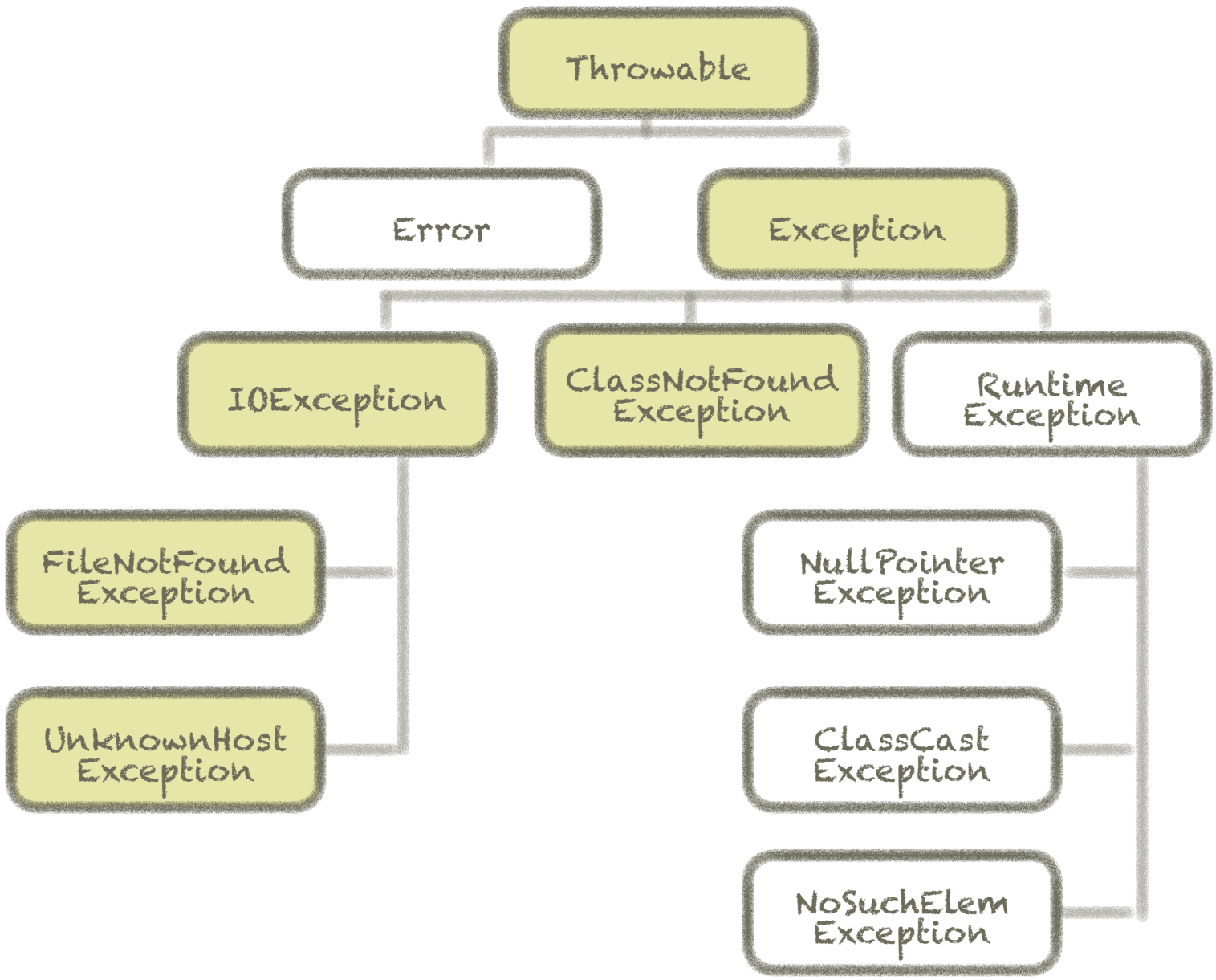
# Exceptions: afhandelen

```
public void method() {
    // do stuff...

    if (...) // something goes wrong
        throw new RuntimeException("AAH!");

    // other stuff...
}

public static void main(String[] args)
{
    try {
        method();
    } catch (RuntimeException e) {
        System.out.println("something went wrong. Ex: " + e);
    }
}
```

Hier zie je hoe je een exception afvangt:

- Je maakt een code block (met accolades) om de code waar de exception gegooid wordt
- Je zet voor het soort exception dat je af wilt vangen een catch blok. Dit blok werkt als een soort methode, die aan wordt geroepen met de gegooide Exception als argument (e). Dit object kun je dus gebruiken om informatie te krijgen over het probleem dat opgetreden is.

# Checked Exceptions




Exceptions zijn objecten zoals ieder ander. Je kunt ze aanmaken, extenden, etc. Hier zie je de belangrijkste typen.

- Error wordt alleen gebruikt voor serieuze fouten waar een normale applicatie geen rekening mee hoeft te houden.
- De witte groene exceptions zijn **checked**. Dit betekent dat als een je ze gooit, je moet declareren dat je ze gooit en als een methode die je aanroept een Exception declareert, dan moet je daar iets mee doen.

# Checked Exceptions

```
/**
 * Reads the contents of a file into a string
 */
public String fileToString(File file)
    throws FileNotFoundException
{
    if (! file.exists())
        throw new FileNotFoundException(
                                "File " + file + "not found");
        ...
}
public void outer()
{
    // ask user for file
    File file = ...;
    String contents = fileToString(file);
}
```

Deze code compileert niet. De laatste regel kan een FileNotFoundException opleveren. Deze exception is checked, dus moet de programmeur er iets mee doen. Er zijn twee opties.

# Checked Exceptions afhandelen

```
public void outer()
{
    boolean fileRead = false;
    while (! fileRead)
    {
        File file = ...; // ask user for file
        try {
            String contents = fileToString(file);
            fileRead = true;
        } catch (FileNotFoundException e)
        {
            System.out.println("Not found, try again");
        }
    }
}
```

Hier zien we een voobeeld van hoe je een exception af kunt handelen. We gebruiken wederom een try/catch blok en we geven de gebruiker een foutmelding als het fout gaat. Door de loop blijft het programma om input vragen totdat er een correcte file ingevoerd wordt.

# Checked Exceptions doorgeven

```
public void outer()
    throws FileNotFoundException
{
    File file = ...;
    String contents = fileToString(file);
}

public void outer()
    throws IOException
{
    File file = ...;
    String contents = fileToString(file);
}
```

Als de methode waar je inzit de checked exception zelf ook gooit, dan hoef je niets af te handelen. Dit mag zelfs als de methode een superklasse gooit van de Exception die je tegenkomt. In de tweede methode is IOException een superklasse van de FileNotFoundException die we tegenkomen. We hoeven die dus niet af te vangen.

# Meerdere Exceptions

```
Database db = ...;

try {
    db.open();
    db.write(information);
    db.close();
} catch (DatabaseUnreachableException e) {
    System.out.println("Database connection lost.");
    ...
} catch (DatabaseLockedException e) {
    System.out.println("Database locked by other user.");
    ...
} catch (DatabaseException e) {  // superklasse
    System.out.println("Unknown database problem.");
    ...
} finally {
    db.close();
}
```

Je kunt ook meerdere Exceptions afvangen voor een gegeven use case. Hier zie je een voorbeeld. Voor specifieke exceptions kunnen we melden wat er fout is gegaan een dat op een speciale manier afhandelen. Als we alleen weten dat het een DatabaseException is (maar niet welke DatabaseException) kunnen we op zijn minst nog zeggen dat er iets is fout gegaan met de database.

We zien hier ook het finally keyword. De code in dit blok wordt altijd uitgevoerd. Of er nou een exception is opgetreden of niet. In dit geval willen we de database connectie altijd netjes afsluiten, want een openstaande connectie is een security en performance issue.

# Tips

Geef informatie: wat was er fout, wat waren de waarden van de variabelen

```
throw new IllegalArgumentExeption("Age ("+age+") can’t be negative");
```

Gebruik exceptions alleen voor onverwacht gedrag

Wees voorzichtig met afhandelen. Je ontneemt de hogere processen de toegang tot de informatie dat er iets fout is gegaan. Doorgeven is meestal veiliger.

# Wat als ik van een checked Exception af wil?

```
// nooit, nooit doen
try {
    s = readFile(file);
} catch (FileNotFoundException e)
{
    // I don't care
}

// Betere oplossing
try {
    s = readFile(file);
} catch (FileNotFoundException e)
{
    // I don't care
    throw new RuntimeException(e, "Something went wrong");
}
```

Soms dwingt code je om iets met een exception te doen, terwijl je geen zin hebt om hem helemaal naar boven door te geven. Het is levensgevaarlijk om de Exception te negeren. Hiermee ontneem je de hogere code iedere mogelijkheid om te merken dat er iets fout is gegaan. De gemaakt String is nu null en gaat ergens in de toekomst, ver van deze code een ingewikkelde bug veroorzaken.

Het tweede voorbeeld geeft een betere oplossing. RuntimeException heeft een constructor die een andere Throwable accepteert en in de RuntimeException stopt. De RuntimeException wordt nu doorgegeven, het programma crasht nog steeds, met een verwijzing naar de juiste code. Het mooiste is, java print de stacktraces van alle Exceptions uit, eerst de RuntimeException, en dan de FileNotFoundException, dus je verliest geen informatie.

# Samenvatting

Java helpt je niet om vlot te programmeren, het helpt je om **veilig** te programmeren.

3 principes van veilig programmeren?

Generics: voor klassen met speciale relaties tot andere klassen, bijv. `List<String>`

Exceptions: voor als er iets fout gaat

Checked exceptions: opvangen of doorgeven

vragen?

# things to try

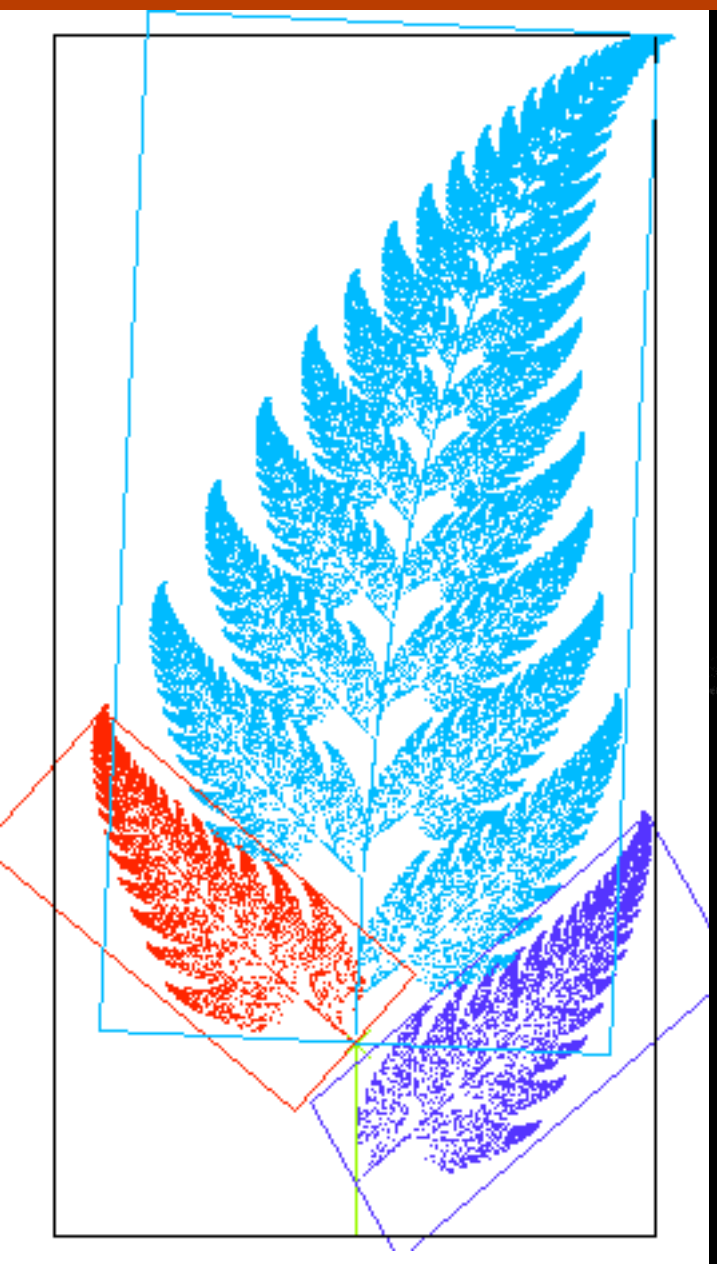
```
int width = 600;
int height = 400;
BufferedImage im =
    new BufferedImage(
                    width, height, BufferedImage.TYPE_INT_ARGB);

// loop over all pixels
for (int i  = 0; i < width; i++)
    for (int j = 0; j < height; j++)
        im.setRGB(i, j, new Color(0.1f, 0.2f, 0.4f).getRGB());

// write a PNG
ImageIO.write(im, "PNG", new File("myimage.png"));
```

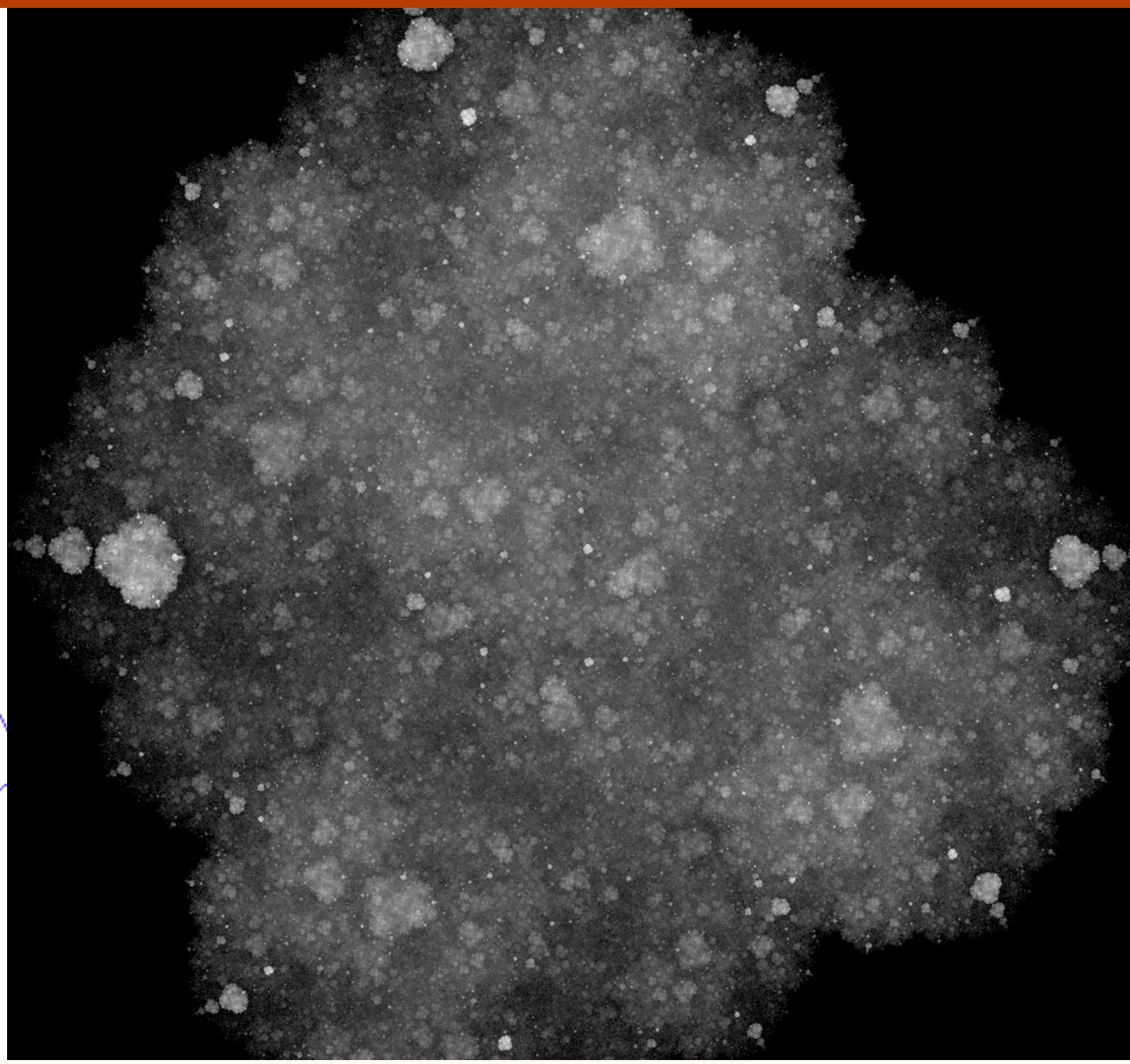
volledige versie
https://gist.github.com/pbloem/a15acc128da3d085f3d9

We sluiten weer af met wat dingen om te proberen. Met deze code kun je een plaatje genereren. Het plaatje is niets meer dan een smaakvol donkerblauw vlak, maar door de kleuren op de juiste manier te varieren, kun je al heel wat bereiken.

Met wat simpele aanpasingen kun je zelfs een sequentie van plaatjes genereren. Die sequentie kun je dan weer met bijvoorbeeld VirtualDub, AviDemux of FFMpegX aan elkaar plakken tot een filmpje.

# 1) Iterated Function Systems



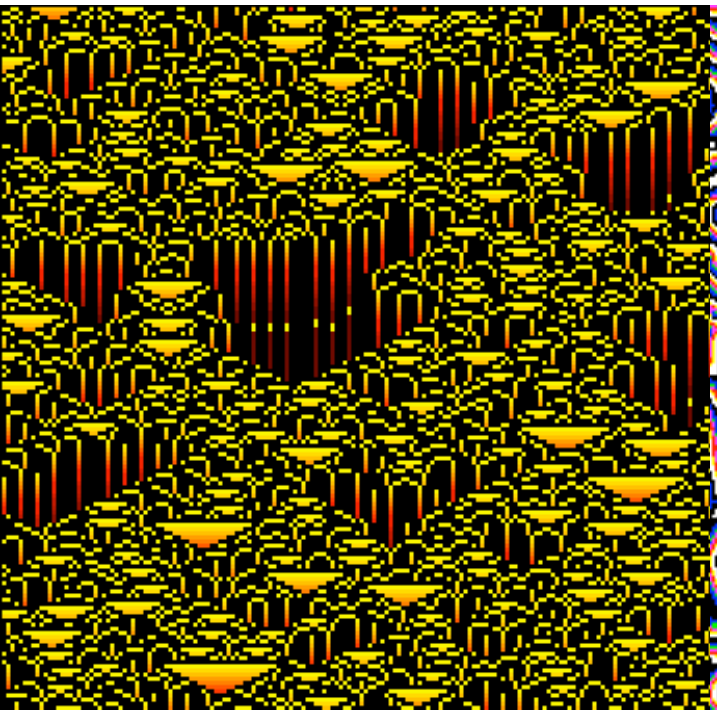
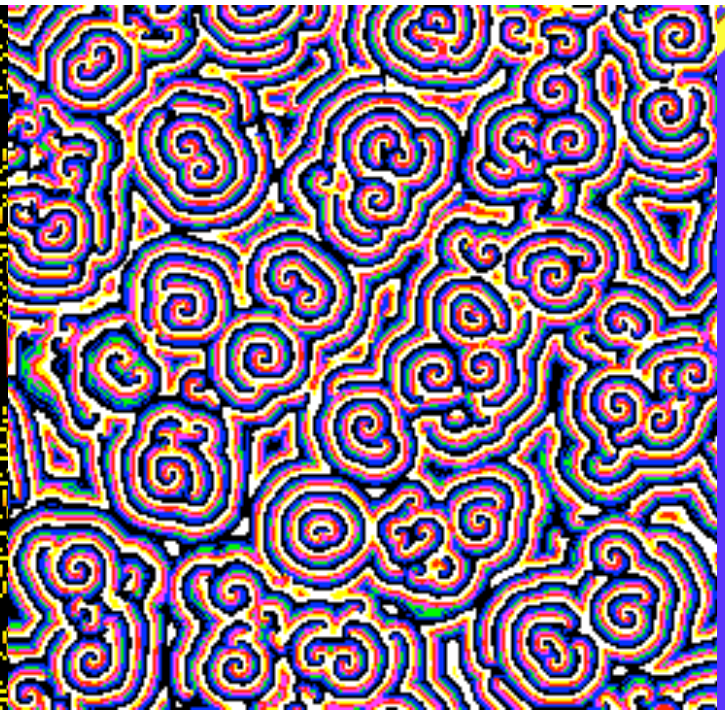
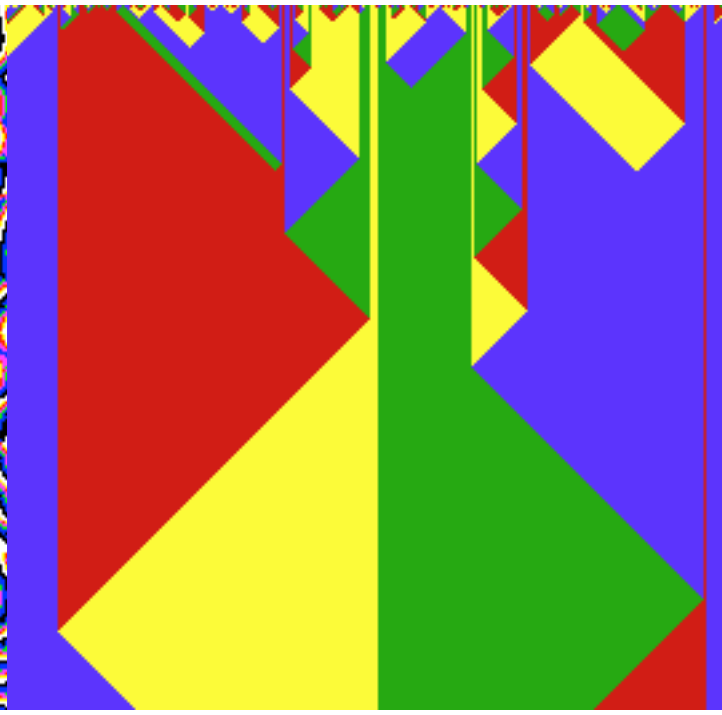


Definieer drie (of meer) functies in het vlak. De input is een x en een y coordinaat en de output ook. Zorg dat de drie functies contractief zijn, dwz. de afstand tussen twee punten wordt altijd kleiner nadat ze beiden door de functie gehaald zijn. Begin met een willekeurig punt. Kies willekeurig een van de drie functies en pas hem toe. Herhaal dit proces en plot alle punten die je tegenkomt. Het resultaat is een Iterated fucntion System. Afhankelijk van hoe je de functies kiest krijg je de meest wonderlijke figuren.

Een goed beginpunt is om drie hoekpunten van een driehoek te nemen en de drie functies zo te definieren dat de output 50% dichter bij een van de drie hoekpunten ligt. De resulterende figuur is een Sierpinski Gasket.

De rechter figuur is een Fractal Flame; een ver gevorderde vorm van IFS.

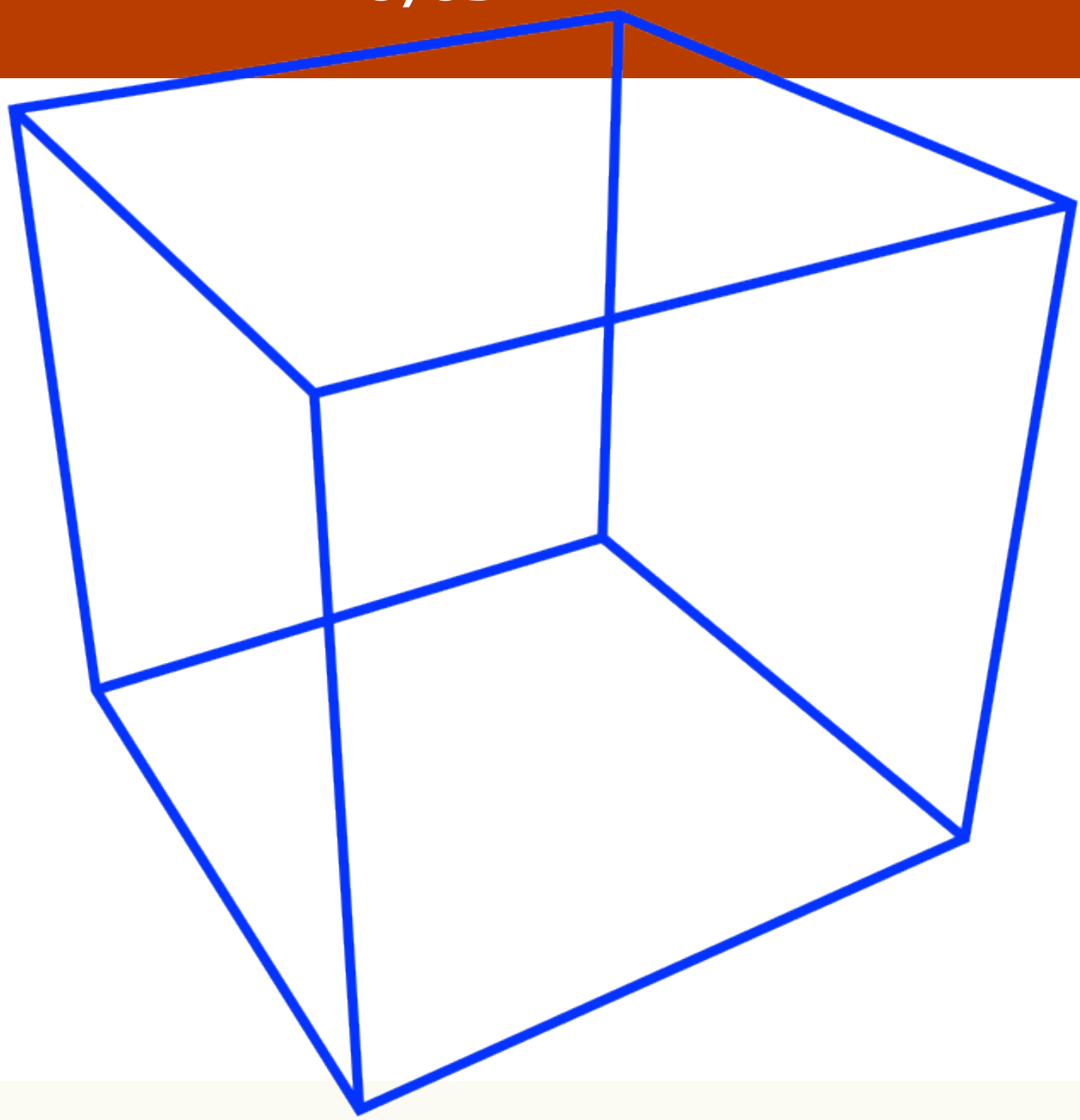
# 2) Cellulaire Automaten

Een cellulaire automaat is een tabel van nullen en enen. De tabel wordt in een loopje ge-update. De waarde van een cel in een volgende generatie is afhankelijk van de waarden van zijn buren in de huidige generatie. De precieze afhankelijkheid is het “programma” van de automaat. Ieder programma geeft andere vormen en structuren.

Het is niet moeilijk om het concept uit te breiden naar bijvoorbeeld grijswaarden en kleuren.

# 3) 3D



Een 3D kubus tekenen is redelijk simpel. Stel je de afbeelding voor als een vlak in de 3D ruimte. Vlak achter het vlak ligt een brandpunt en voor het vlak ligt de kubus. Voor ieder hoekpunt trekken we een lijn van het punt naar het brandpunt. Waar deze lijn de afbeelding snijdt is met middelbare school wiskunde te vinden. Verder hoef je alleen maar lijnen te trekken tussen deze punten in de afbeelding als er een lijn tussen de punten staat in de kubus.

Als je geen zin hebt om zelf pixel voor pixel een lijn te trekken kun je `createGraphics()` aanroepen op je `BufferedImage`. Je krijgt dan een `Graphics2D` object waarmee je bijvoorbeeld lijnen kunt tekenen, maar ook cirkels, rechthoeken, tekst, etc.

`http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html`