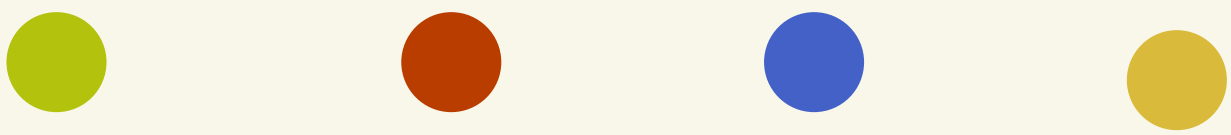
```java
interface Foo { }
class A implements Foo { }
class B extends A { }

A a = new A();
B b = new B();

System.out.println(a instanceof Foo);     // A
System.out.println(b instanceof Object); // B
System.out.println(b instanceof Foo);    // C
```



A B C geen

```java
class Cat { }
class Dog { }

public static void main(String [] args) {

    // Altijd false, compileert
    System.out.println(null instanceof String)

    // True
    int [] nums = new int[3];
    System.out.println(nums instanceof Object);

    // Altijd false, compileert niet
    Integer i = new Integer(1);
    System.out.println(i instanceof Double);
}
```

Nog een aantal interessante details over instanceof.

Je mag `instanceof` doen op een null. Dit is altijd false.

Arrays zijn technisch gezien objecten. Ze hebben geen mooie toString of equals methoden, maar ze hebben ze wel.

Als je een vergelijking maakt van twee types die nooit een instanceof relatie kunnen hebben, dan zal je code niet compileren.

```java
public static void main(String[] args)
{
    try {
        File file = new File("myfile.txt");
        FileReader reader = new FileReader(file);
    } catch (IOException e) {
            System.out.println("IOException");
    } catch (FileNotFoundException e) {
            System.out.println("FileNotFoundException");
    }

}
```



“A” geen output “B”

```java
class Polygon {
    int numSides;
    public Polygon(int numSides, Point... corners) {
        this.numSides = numSides;
    }
}

class Triangle extends Polygon {
    double area;
    public Triangle(Point c1, Point c2, Point c3){
        area = computeArea(c1, c2, c3);
        super(3);
        ...}

    public double computeArea(Point c1, Point c2, Point c3) {...}
}
```

Compileert niet

Exception

Hoe schrijf ik BRUIKBARE code?

java@peterbloem.nl

Wat gebeurt er in je computer als je in Youtube een filmpje bekijkt?

De frames worden gerenderd door een Flash object, dat aangemaakt is door de rendering engine, die aangeroepen wordt door je browser. De browser tekent zijn elementen via commando's aan het operating system. De browser en het operating system zijn geschreven in C of een variant. Dit wordt gecompileerd naar machine-taal, en de machinetaal wordt aan de processor gevoerd. In de processor worden de x86-64 instructies vertaald naar micro-operaties en uitgevoerd.

Ook tussen het flash object en de webpagina zit een communicatielaag en tussen de webpagina en de YouTube server, en tussen het flash object en de YouTube server. De hele ICT wereld is een sterrenstelsel van *interfaces*.

Zie ook http://xkcd.com/676/

Abstracties en interfaces

machinetaal → assembly

assembly → C

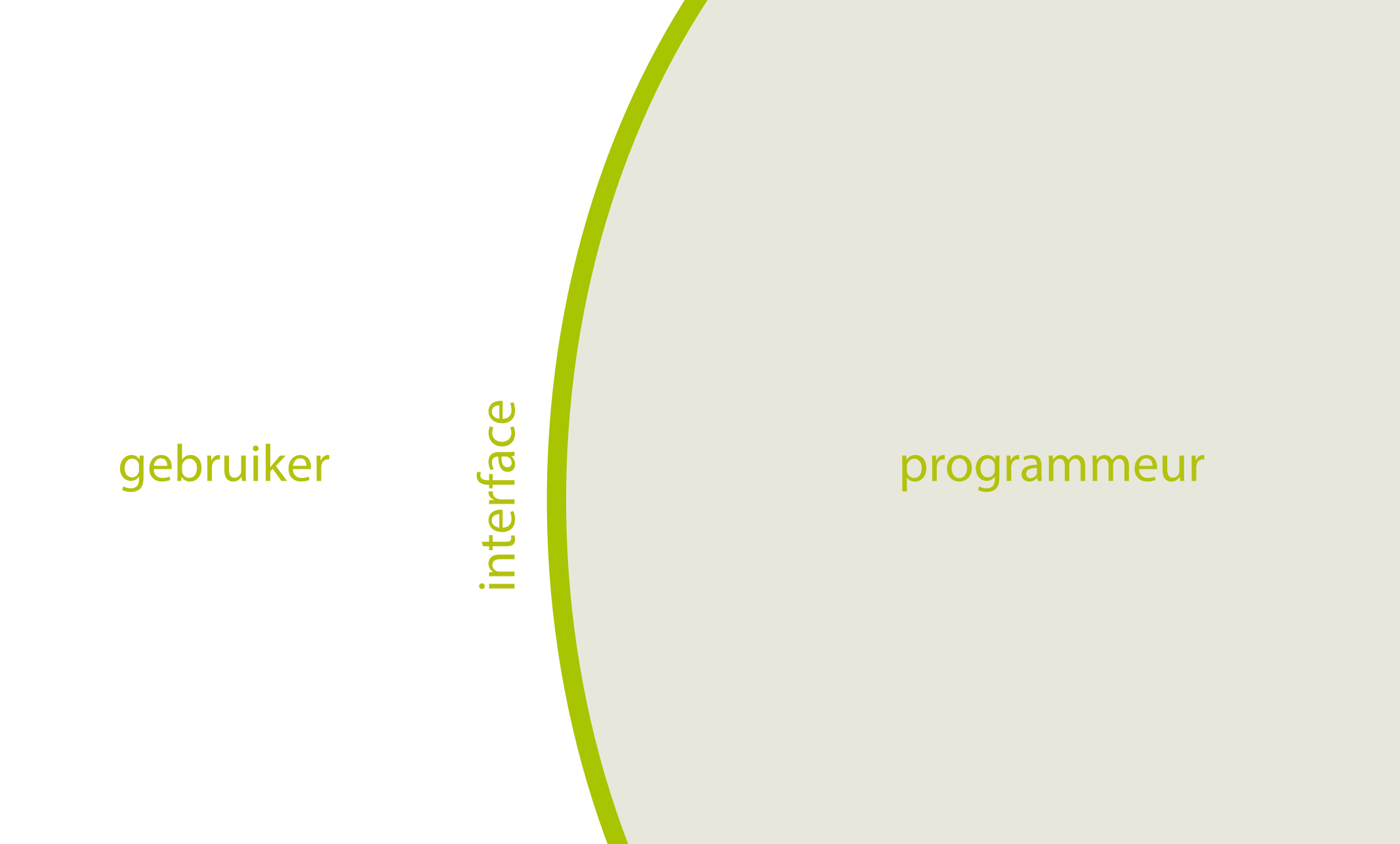
C → C++

C++ → Java

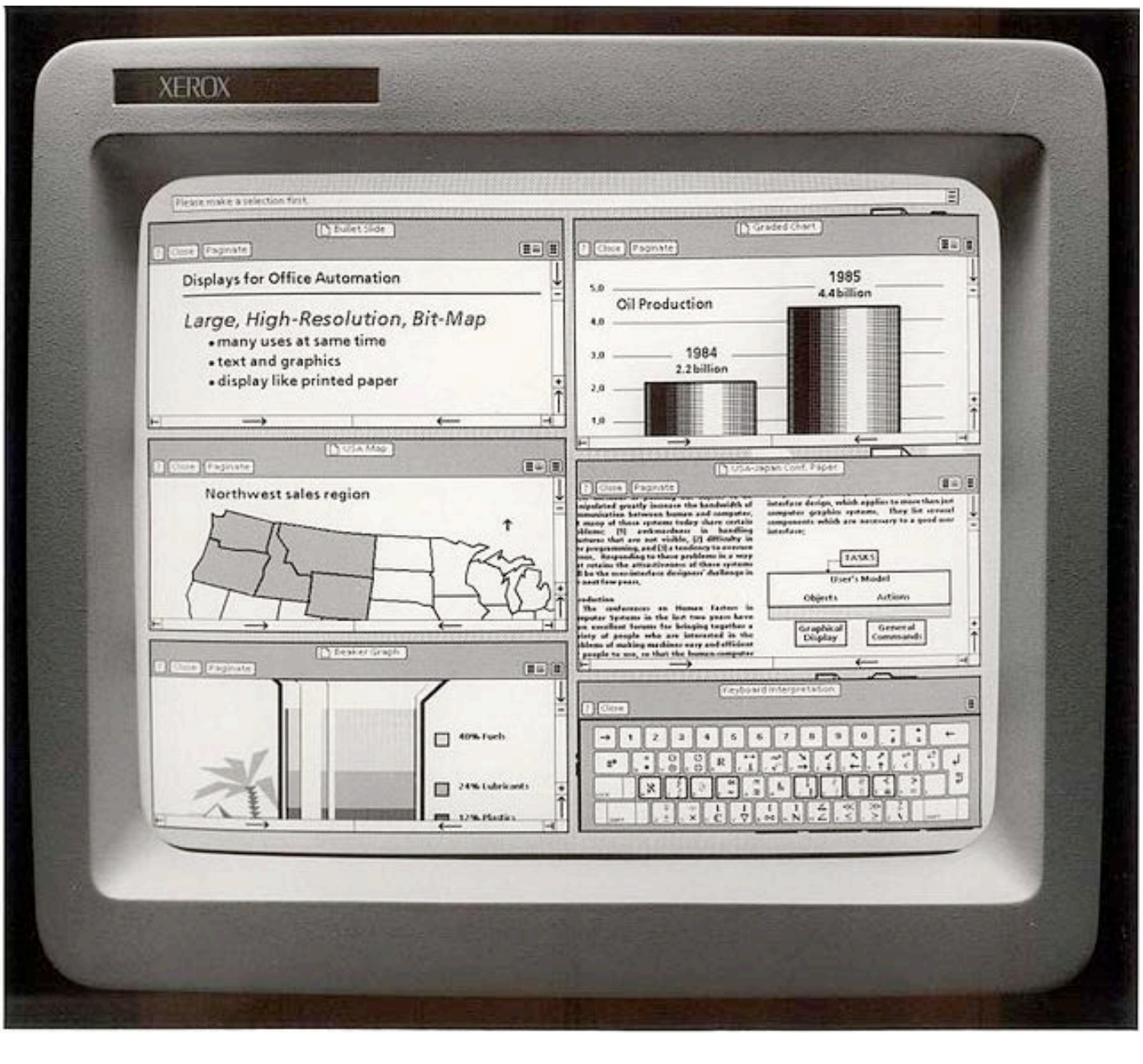
Java → Scala

“Separation of concerns”

Het maken van een goede abstractie komt neer op “separation of concerns”. Een programmeur wil zich met één aspect van zijn code tegelijk bezighouden. Een C++ programmeur bijvoorbeeld, moet zorgen dat zijn programma klopt en dat het zijn geheugen correct gebruikt. Java abstraheert een van deze concerns naar een garbage collector zodat de programmeur zich daar geen zorgen over hoeft te maken.



Dit diagram geeft het principe van een goede interface aan. Er is een gebruiker, die iets wil bereiken, en een programmeur die dat mogelijk moet maken. Het is de taak van de programmeur om zoveel mogelijk complexiteit te verbergen voor de gebruiker, en net genoeg complexiteit over te laten dat de gebruiker zijn probleem op kan lossen.



Het simpelste voorbeeld is de GUI. Hier haalt de programmeur bijna alle complexiteit van computerhardware weg. De gebruiker hoeft niet te weten hoe een programma er uit ziet van binnen, wat computergeheugen is, of wat er in zit. Alleen de concepten die voor de gebruiker van belang zijn (files, “de harde schijf”, de monitor) worden doorgegeven (op een versimpelde manier).

Daarnaast bouwt de programmeur verschillende nieuwe concepten (windows, iconen, menu’s) om de oude concepten te versimpelen (files, commando’s, processen).

Maar ook op lagere niveau's zien we dit soort abstracties. Neem bijvoorbeeld een Array in java. Arrays zijn moeilijk te gebruiken. Je moet van te voren weten hoe groot ze zijn en als je dat niet weet moet je ze telkens overhevelen naar een nieuw array als het oude te klein is.

Een ArrayList neemt deze complexiteit weg. De ArrayList vormt een interface om het array heen. Iedere keer als je iets aan een ArrayList toevoegt, controleert de ArrayList of het interne array nog groot genoeg is, en vergroot het als dat niet zo is. De gebruiker merkt hier niets van en kan zich richten op andere dingen.

```java
/**
 * Reverses the order of the elements in the specified list.<p>
 * This method runs in linear time.
 * @param  list the list whose elements are to be reversed.
 * @throws UnsupportedOperationException if the specified list or
 *         its list-iterator does not support the <tt>set</tt> operation.
 */
public static void reverse(List<?> list) {
        int size = list.size();
        // Check whether the list can be accessed randomly
        if (size < REVERSE_THRESHOLD || list instanceof RandomAccess) {
            for (int i=0, mid=size>>1, j=size-1; i<mid; i++, j--)
                swap(list, i, j);
        } else {
            ListIterator fwd = list.listIterator();
            ListIterator rev = list.listIterator(size);
            for (int i=0, mid=list.size()>>1; i<mid; i++) {
                Object tmp = fwd.next();
                fwd.set(rev.previous());
                rev.set(tmp);
            }
        }
}
```

Ook op het niveau van methodes zie je dit principe terug. Een methode heeft een binnenkant (grijs) en een buitenkant (blauw). De gebruiker moet aan de blauwe tekst: de signature en het javadoc commentaar kunnen zoen wat de methode doet en hoe hij gebruikt moet worden.

Als de gebruiker in de methode (de grijze tekst) moet kijken om hem correct te kunnen gebruiken is de methode fout opgezet.

Dit geldt ook voor de methoden die je alleen voor jezelf schrijft. Als je je methoden op deze manier opzet, kun je later, wanneer je de methode aanroept, de implementatie vergeten en je richten op belangrijker dingen.

mind your interfaces

Waarom?

1. Interfaces zijn moeilijk te veranderen
2. Interfaces zijn zichtbaar
3. Goede interfaces dwingen simpliciteit, modulariteit en hergebruik af
4. Helpt om keuzes te maken
5. Goed programmeren is interfaces ontwerpen

API's

Application Programming Interface

verzameling van interfaces (evt met implementatie):

1. Java Collections
2. Twitter API
3. Scanner

Een library met de focus op de interfaces

Wat maakt een API goed?

1. Makkelijk te gebruiken
2. Moeilijk verkeerd te gebruiken
3. Simpele use-cases simpel op te lossen
4. Moeilijke use-cases op te lossen
5. Goeie power-to-weight ratio

Hoe ontwerp ik een API?

Werk van buiten naar binnen:

1. Wie is de gebruiker, wat wil de gebruiker?
2. User stories
3. Use cases
4. Eerst code de de API gebruikt, dan code die de API implementeert

Use Cases, User Stories en Requirements

“Er moet een querytaal komen om objectcollecties samen te kunnen stellen.”

“Ik wil alle gebruikers opvragen met leeftijden tussen 17 en 25.”

“Bij het afsluiten moet het programma vragen of het document opgeslagen moet worden als er veranderingen zijn gemaakt”

“Ik wil zekerheid hebben dat ik geen werk kwijtraak.”

“Serverconnecties moeten kunnen worden afgesloten.”

“Ik wil geen open connecties overhouden.”

Hier wat voorbeelden van user stories (soms ook use cases of requirements genoemd, het verschil is niet zo belangrijk).

De rode zinnen zijn geen goede use cases. Ze leggen te veel vast wat de oplossing moet zijn. Hiermee ontnemen ze de vrijheid van de ontwerper om op zoek te gaan naar een elegante oplossing die meerde use cases tegelijk mogelijk maakt. Het legt ook de verantwoordelijkheid voor het vinden van een oplossing bij de gebruiker in plaats van de ontwerper. Een goede use case legt alleen vast wat de gebruiker wil. Hoe dat opgelost moet worden, wordt pas uitgezocht als alle user stories verzameld zijn.

De middelste use case is een goed voorbeeld. Een dialoogvenstertje dat vraagt of je je werk wilt opslaan is in dit geval een slechte oplossing. Het zou mooier zijn om bijvoorbeeld automatisch op te slaan, in combinatie met een goede versiegeschiedenis. Daarmee wordt ook voldaan aan de use case “ik wil terug kunnen naar een oude versie van mijn document”.

3 kenmerken van een slechte API

1) Implementation leaks

```
public static int calculatePension(Person person, Pension plan)
    throws SQLException
{...}

//---

List<String> list = ...;
Sorter sorter = new new BubbleSorter(list);
sorter.sort();

//---
/**
 * Retrieve information from the webpage for a given person
 */
public static DOMTree retrieve(Person person)
{...}
```

Een implementation leak is de situatie waar je de details van je gekozen implementatie-door laat schijnen aan de gebruiker. Dit veroorzaakt twee problemen.

In de eerste plaats zorgt het ervoor dat dingen voor de gebruiker moeilijker worden. In het eerste voorbeeld geeft de calculatePension methode een SQLException terug (SQL is een taal om databases aan te spreken). Dat de gegevens uit een database komen kan verborgen blijven voor de gebruiker. De gebruiker hoeft zelfs niet eens te weten wat SQL is.

Ten tweede wordt alles wat je via de interface aan de gebruiker laat zien vastgelegd. Als je in de toekomst de gegevens niet meer in een SQL Database wilt hebben, maar in iets anders (een NoSQL database bijvoorbeeld), zit je nog steeds vast aan de SQLExceptions.

In het tweede voorbeeld moet de gebruiker zelf de keuze maken welk sorteeralgoritme gekozen moet worden. Een betere oplossing zou zijn om een goede standaardkeuze te maken, en de gebruiker alleen lastig te vallen met de details als dat nodig is.

2) Slechte namen

```
Font, Set, PrivateKey, Lock,
ThreadFactory, TimeUnit

DynAnyFactoryOperations,
_BindingIteratorImplBase, ENCODING_
CDR_ENCAPS, OMGVMCID
```

Het kiezen van een juiste naam is ontzettend belangrijk. Je API is een soort taal bovenop Java. Je gebruiker moet die taal leren.

Namen moeten “self-explanatory” zijn, kort en consistent (gebruik niet de ene keer delete en de andere keer remove).

Als je geen goede naam kunt bedenken, is dat meestal een teken dat je teveel functionaliteit op één element probeert te laden. Denk dus niet dat naamgeving niet zo belangrijk is en dat er betere dingen zijn om je tijd aan te besteden. Een slechte naam betekent dat er meer fout is, en zadelt de gebruiker op met een ondoorgrondbare API.

(Deze namen komen uit de presentatie *How To Design a Good API and Why it Matters* van Joshua Bloch : `https://www.youtube.com/watch?v=aAb7hSCtvGw`)

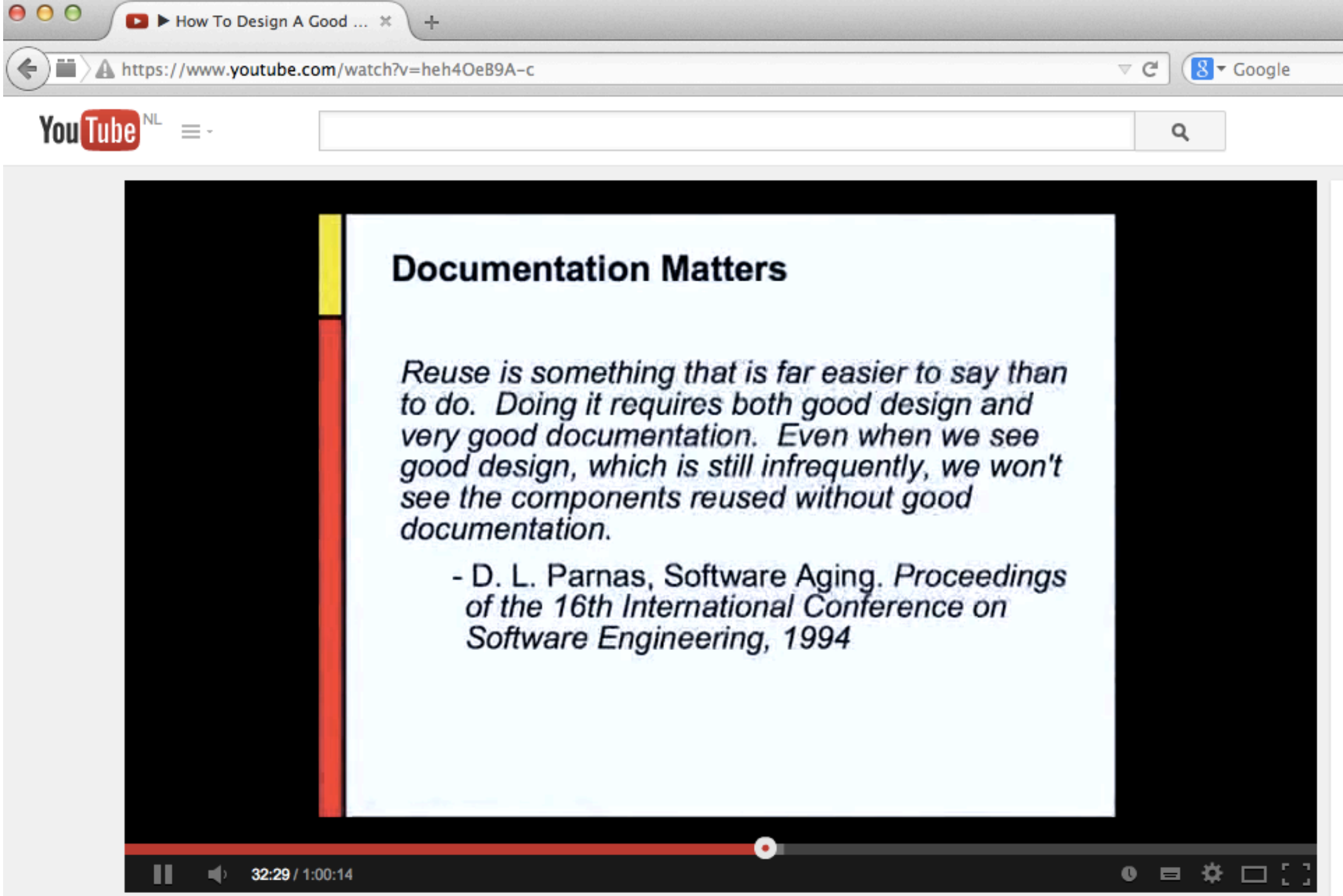
3) Boilerplate code

```
EnvironmentConfig environmentConfig = new EnvironmentConfig();
environmentConfig.setAllowCreate(true);

// perform other environment configurations
File file = new File(DATA_DB_DIR_NAME);
Environment environment = new Environment(file,
          environmentConfig);
DatabaseConfig databaseConfig = new DatabaseConfig();
databaseConfig.setAllowCreate(true);

// perform other database configurations
_sleepyCatDB = environment.openDatabase(null, DB_NAME,
            databaseConfig);
```

BerkeleyDB is een soort collectie (list, map, etc) die dingen persistent opslaat. Dwz, ze worden naar de harde schrijf geschreven, zodat je je java programma af kunt sluiten, opnieuw op kunt starten en je objecten weer kunt gebruiken. Dit is de code die nodig is om een nieuwe database te maken.

Dit zou in één regel moeten kunnen, maar er zijn er 7 nodig. Ik moet niet alleen een Environment, maar ook een EnvironmentConfig maken. Dit zijn vast nuttige objecten, maar ik heb een simpele use case. Pas als ik een ingewikkelde use case heb waar een EnvironmentConfig nutig is, wil ik hem tegenkomen.

Boilerplate code is niet alleen frustrerend, het is ook gevaarlijk. In plaats van de code zelf te typen, kopieer je hem en pas je kleine dingen aan. Dit werkt bugs in de hand.



De meeste dingen die we hier verteld hebben komen uit deze lezing van Joshua Bloch.

Na de pauze lopen we door de features van Java heen die we nog over hebben

Packages

```java
import java.util.List;
import java.util.ArrayList;

public class Blorg {
    ...
}
```

```java
package nl.peterbloem.util;

public class Utilities {
    ...
}
```

Packages zijn een manier om in Java je code te organiseren. Een naam als List of Map is zo nuttig, die wil je vaker kunnen gebruiken. Daarom zet je objecten die samen een collectief vormen bij elkaar in een package. Met een import statement importeer je de juiste versie; bijvoorbeeld java.util.List.

Als je zelf grotere softwareprojecten gaat schrijven zul je je eigen code ook willen organiseren in packages. Dit doe je met het package keyword. Daarachter plaats je een reeks alfanumerieke karakters, gecheiden door punten. De conventie is om de hostname van je organisatie omgekeerd te gebruiken. Java vereist dat de mappenstructuur overeenkomt met de package strctuur. Deze klasse moet dus in de directory `./nl/peterbloem/util` staan.

Reguliere expressies

`harmoni[sz]e`	harmonise, harmonize
`\d\d\d`	123, 456, 000, 007
`\d*`	"", 2, 43, 566, 983
`\d\d(\s*)\d\d\d`	22355, 34 345, 37 485
`\[.*\]`	[a], [bol], [wlfhwsf], []

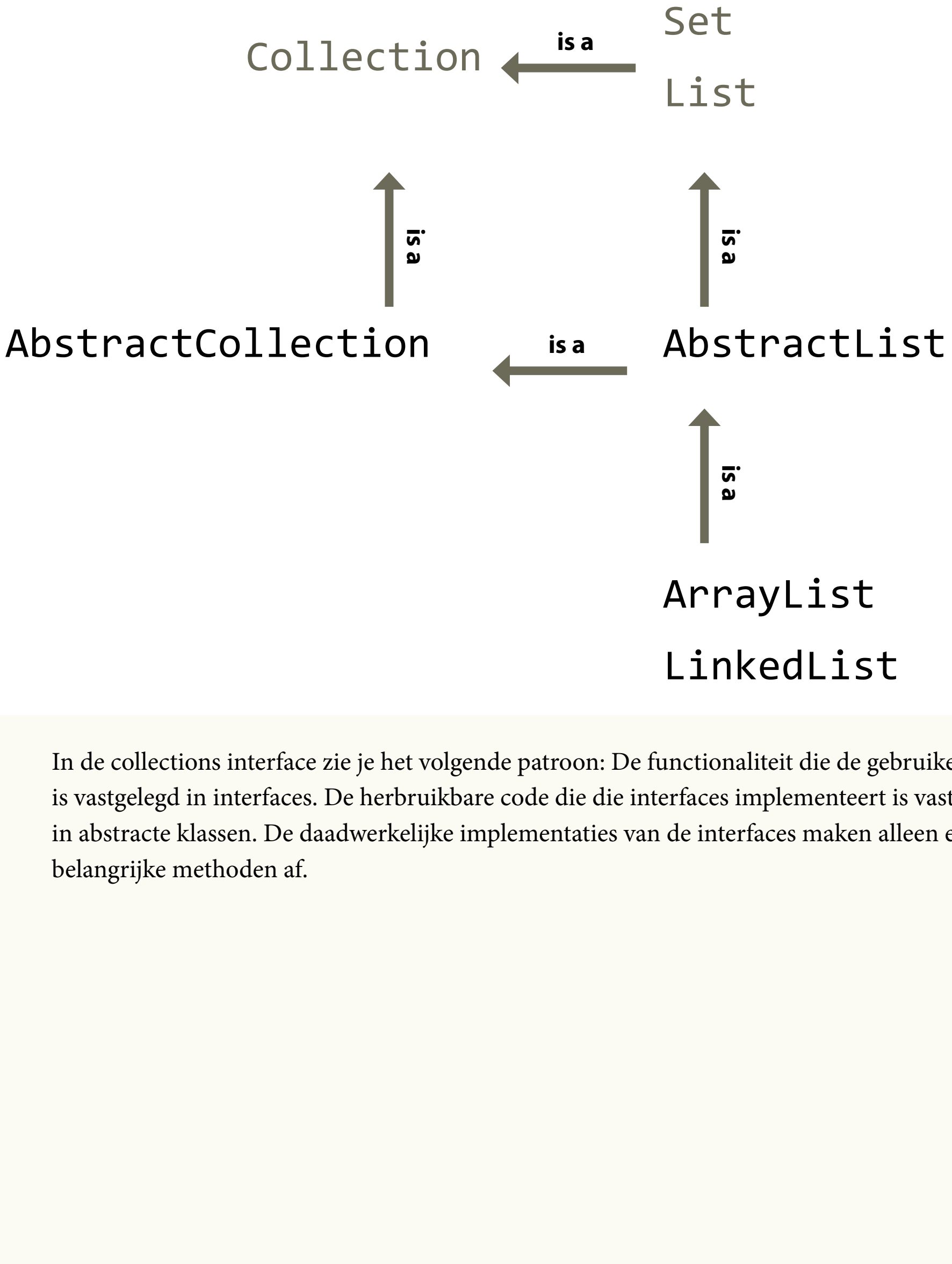
Reguliere expressies zijn een manier om een verzameling van strings te definiëren. Dit wordt vaak gebruikt om in teksten te zoeken, input te valideren, of teksten te manipuleren.

Reguliere expressies zijn vaak een slechte oplossing voor een probleem. Ze zijn moeilijk te lezen, moeilijk aan te passen door iemand die jouw code aan moet passen en zelfs als het lijkt te werken doet het vaak net iets anders dan je dacht. Wees er dus voorzichtig mee, maar soms is het de beste oplossing.

Reguliere expressies

```
String[] split = "a, b,c;   d,  e".split("[,;]\\s*");

// resultaat: {"a", "b", "c", "d", "e"}

"06--678".matches("\\d\\d[-]*\\d\\d\\d");

// resultaat: true

Pattern pattern = Pattern.compile("(\\w*)@(\\w.\\w)");
Matcher m = pattern.matches("java@peterbloem.nl");
if(m.matches())
{
    for (int i = 0; i < m.groupCount(); i++)
        System.out.println(m.group(i));
}
```

Collections, nog een keer



In de collections interface zie je het volgende patroon: De functionaliteit die de gebruiker ziet is vastgelegd in interfaces. De herbruikbare code die die interfaces implementeert is vastgelegd in abstracte klassen. De daadwerkelijke implementaties van de interfaces maken alleen een paar belangrijke methoden af.

```
public interface List<E> extends Collection<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
    List<E> subList(int fromIndex, int toIndex);
}
```

Dit is de List interface. Het kan heel nuttig zijn om een object deze interface te laten implementeren, maar zoals je ziet moet je er een boel werk voor doen.

Zelf een List maken

```
public MyStamps extends AbstractList<Stamp>
{
    @Override
    public int size() {
        return numStamps;
    }

    @Override
    public Stamp get(int i)
    {
        return ...;
    }
}
```

Gelukkig kun je de AbstractList klasse extenden. Deze implementeert alle methoden van de Lijst interface aan de hand van de twee methoden size() en get(int i). Dit zijn abstracte methoden, die je in je implementatie nog moet implementeren. Met deze twee methoden heb je een werkende List.

Voorbeeldje

```
public Range extends AbstractList<Integer>
{
    private final int size;

    public Range(int size){ this.size = size; }

    @Override
    public int size() {
        return size;
    }

    @Override
    public Integer get(int i)
    {
        return new Integer(i);
    }
}
```

Dit is een klein voorbeeld van wat je met dit principe kunt doen. Dit object slaat niks op, maar kan met een enkel integer field een lijst van 1 tot n integers representeren.

Maps

```
Map<Integer, String> users = new HashMap<Integer, String>();

users.put(3, "Peter Bloem");

System.out.println(users.get(3));

// resultaat: Peter Bloem

System.out.println(users.get(0));

// resultaat: null

System.out.println(users.get(0).toLowerCase());

// NullPointerException
```

Een Map is een andere nuttige collectie. Met een Map<K, V> sla je associaties tussen paren van objecten op. Je gebruikt een key van type K en een value van type V. Aan de hand van je keys kun je je values opslaan en weer opvragen.

Als je een key gebruikt om iets op te vragen die de je nog niet eerder gebruikt hebt, krijgt je null terug.

Maps

```
Map<Integer, Set<String>> groups =
    new HashMap<Integer, Set<String>>();
...

// add a user to group 3
if (! groups.containsKey(3))
    groups.put(new HashSet<String>());

groups.get(3).add("Peter Bloem");
```

Dit is een patroon wat je vaak tegenkomt als je meerdere waarden aan een key wilt kunnen hangen. De values zijn van het type set. Als de key nieuw is voeg je een nieuwe set toe en aan die set voeg je je objecten toe.

Autoboxing

```
List<Integer> list = new ArrayList<Integer>();

int x = 13;

list.add(x);
list.add(15);

int y = list.get(0);
```

Java heeft voor iedere primitive (int, double, etc) een *boxed* Object (Integer, Double, etc). Sinds java 1.5 kun je primitives en boxed primitves aan elkaar toewijzen. De compiler converteert automatisch tussen de twee.

```java
public class Test
{
    static void m(Integer x) {
        if(x == 13)
            System.out.println("X");
    }

    public static void main(String[] args) {
        Map<String, Integer> map =
                        new HashMap<String, Integer>();

        map.put("hallo", 13);

        m(map.get("Hallo"));
    }
}
```



“X“	compileert niet	foutmelding

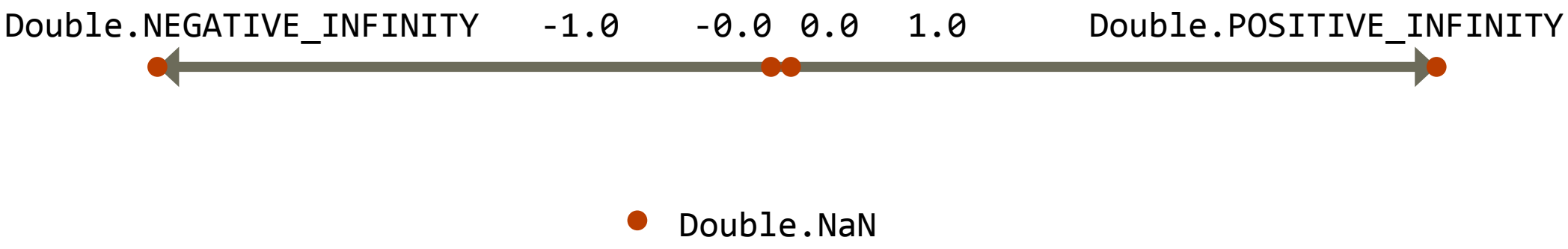
```java
boolean b = true;

Double d1 = 0.13;
Double d2 = null;
Double d = b ? d1.doubleValue() : d2;

System.out.println(d);
```

0.13

null

foutmelding

double (en float)



Als je integers gebruikt heb je het gevaar dat ze overflowen: als de waarde te groot wordt klapt hij om en krijg je een negatief getal. Bij doubles heb je dit probleem niet. Double heeft de standaard getallenlijn uitgebreid met positief en negatief oneindig. Als je waarde te groot wordt of te klein, krijg je oneindig terug als waarde.

Daarnaast heeft Double een waarde NaN (not a number) die je krijgt als je ongedefinieerde operaties uitvoert.

double (en float)

```
System.out.println(1.0 / 0.0);              // Infinity
System.out.println(1.0 /-0.0);              // - Infinity
System.out.println(0.0 / 0.0);              // NaN

System.out.println(Math.pow(2.0, Math.pow(2.0, 100)));
                                             // Infinity


System.out.println(Double.NaN + 1.0);       // NaN
System.out.println(Double.POSITIVE_INFINITY + 1.0);    // Infinity
System.out.println(Double.POSITIVE_INFINITY * -1.0); // -Infinity

System.out.println(0.0/0.0 == Double.NaN);  // false
System.out.println(Double.isNaN(0.0/0.0));  // true
```

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

    Double inc = 0.1;

    for (int i = 0; i < 10; i += 1) {
        System.out.print(“I”);
    }

    System.out.println();

    for (double d = 0.0; d < 1.0; d += inc) {
        System.out.print(“D”);
    }

}
```

gelijk ongelijk NullPointerException

Deze code print een reeks I’s uit en daaronder een reeks D’s. Print hij evenveel I’s als D’s?

Voor 1.5

```
public class BufferedImage {

    ...
    public static final int TYPE_INT_RGB = 1;
    public static final int TYPE_INT_ARGB = 2;

    public BufferedImage(int width, int height, int type) {...}

}

BufferedImage image =
    new BufferedImage(400, 600, BufferedImage.TYPE_INT_RGB);

BufferedImage image =
    new BufferedImage(BufferedImage.TYPE_INT_RGB, 400, 600);
```

Soms heb je een situatie waarbij je een variabele wilt hebben die een handvol waarden kan hebben. Voor java 1.5 gebruikte je daar ints voor. Zoals je in dit voorbeeld ziet kan dat fout gaan.

sinds 1.5: enum

```
public class PokerGame {
    public static enum Suit {SPADES, CLUBS, HEARTS, DIAMONDS}
    ...
}

public PreciseMath {
    public static enum RoundingMode {UP, RANDOM, EVEN, DOWN}
}

public ImageWriter {
    // supported types
    public static enum Type {PNG, JPG, TIFF}
}
```

Sinds 1.5 hebben we enums. Met enums kun je precies het bereik van waarden vastleggen.

enum

```
public class Test
{
    public static enum Suit {SPADES, CLUBS, HEARTS, DIAMONDS}

    public static void main(String args[]) {

        Suit mySuit = Suit.SPADES;
        System.out.println(mySuit); // SPADES

        for(Suit s : Suit.values())
            System.out.print(s + ","); // SPADES, CLUBS, HEARTS, DIAMONDS,

        System.out.println();
        System.out.println(Suit.SPADES.equals(Suit.values()[0]));   // true
        System.out.println(Suit.SPADES == Suit.values()[0]);   // true
        System.out.println(Suit.SPADES.hashCode());    // 460029827
    }
}
```

Enums zijn achter de schermen ge-implementeerd als singleton objecten. Het is een klasse waarvan maar een paar instanties moegen bestaan (één voor iedere mogelijke waarde).

```java
public class Test
{
    public static void main(String args[]) {

        List<String> list = Arrays.asList("a", "b", "c");

        if(list instanceof List<Object>)
            System.out.println("objects");
        else
            System.out.println("something else");
    }
}
```



“objects” “something else” compileert niet

Type erasure

Generics worden alleen tijdens het compileren gebruikt, daarna weggegooid.

Je kunt niet “live” aan een object zien wat zijn generics zijn.

Dit maakt java code backwards compatible, maar het is soms vervelend

exercises

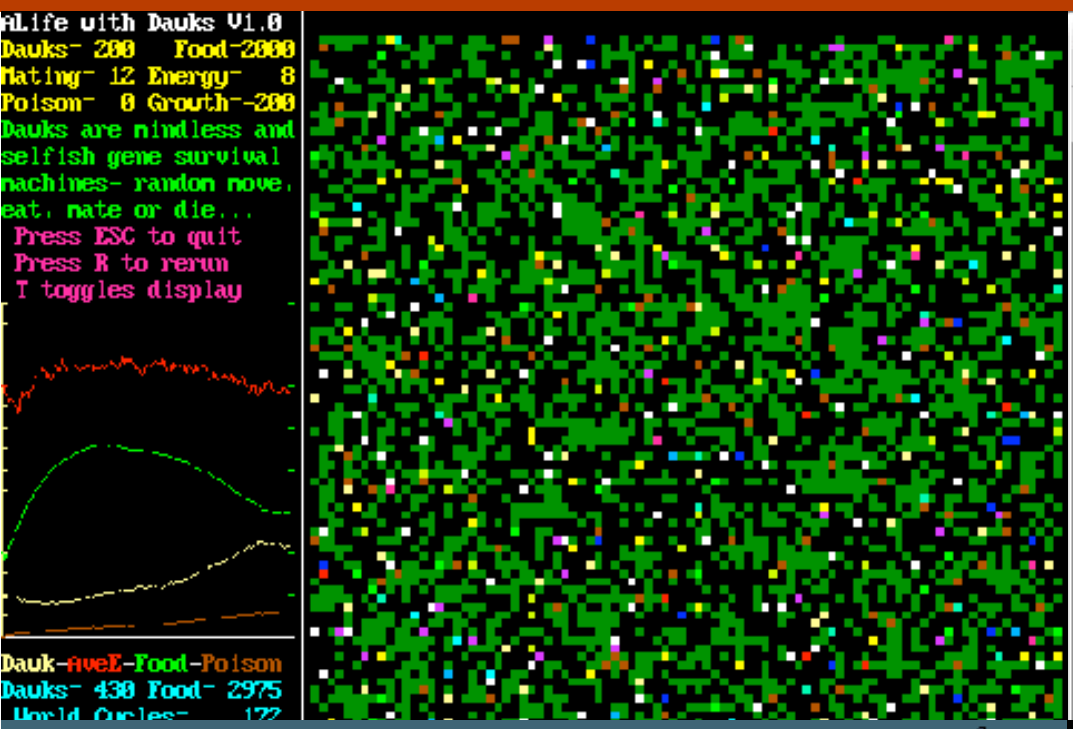
1) muziek

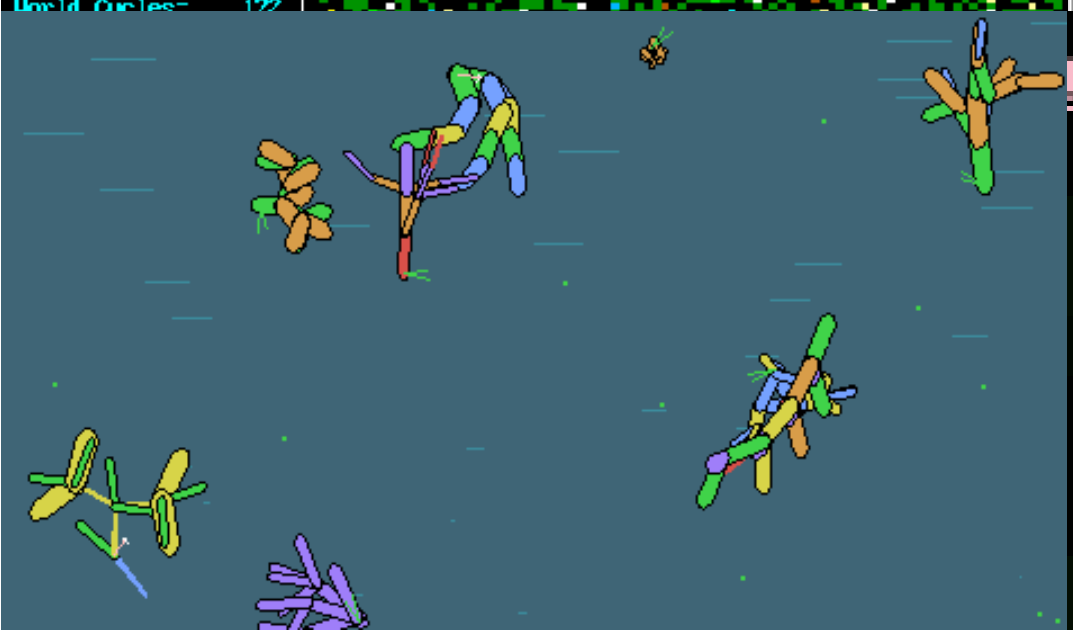
```
private static MidiChannel[] channels;

Synthesizer synth = MidiSystem.getSynthesizer();
synth.open();
channels = synth.getChannels();

// * start playing a note
channels[INSTRUMENT].noteOn(id("6D"), VOLUME);
// * wait
Thread.sleep(600);
// * stop playing a note
channels[INSTRUMENT].noteOff(id(note));
```

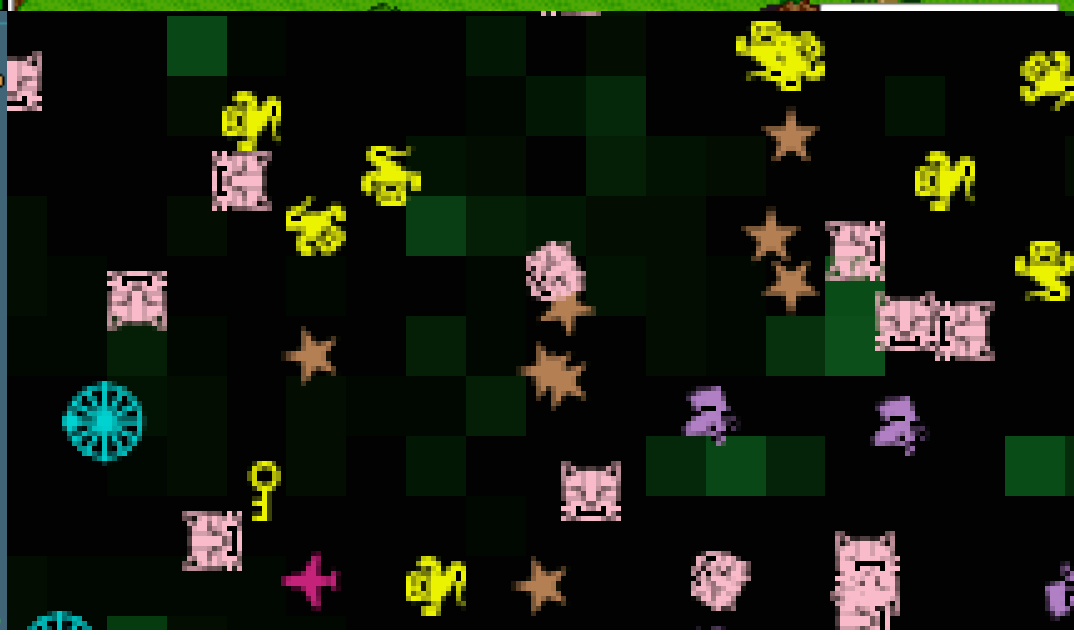
volledige code: **gist.github.com/pbloem** (Synth.java)

Met deze code kun je MIDI muziek afspelen. channels is een array van midi kanalen (dwz. instrumenten). Op ieder kanaal kun je een frequentie aanzetten en weer uitzetten. De id functie in Synth.java kun je gebruiken om een noot te vertalen naar een frequentie.

2) A-life







Artificial life is het idee dat je een simpele versie van een organisme kunt maken in de computer. Je kunt bijvoorbeeld beestjes ter grootte van een pixel maken die op zoek moeten gaan naar voedsel. Je kunt ook ingewikkeldere beestjes maken die eerst moeten leren hun lichaam te bewegen.

3) Network games

```
Socket client =
           new Socket(port);

OutputStream st = kkSocket.getOut-
putStream()
```

```
ServerSocket server =
           new Socket(port);

Socket client = server.accept();

InputStream st =
    clientSocket.getInputStream();
```

http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

Een netwerk client en server maken in Java is relatief simpel. Je moet wat boilerplate code schrijven, maar voor je het weet kun je tussen computers strings versturen. Zodra je dat hebt kun je simpele multiplayer games schrijven. Je kunt bijvoorbeeld een multiplayer text-adventure maken.