Informatica

Les 6

Basis- & slimme algoritmen

Jan Lemeire
Informatica 2e semester
februari – mei 2023

Vandaag

1. GUIs met Java (boek I)
2. Project
3. Iets met functie
4. Interfaces en abstracte klasses
5. Newton’s algoritme
6. Discrete simulaties & Dijkstra
7. Slimme algoritmen

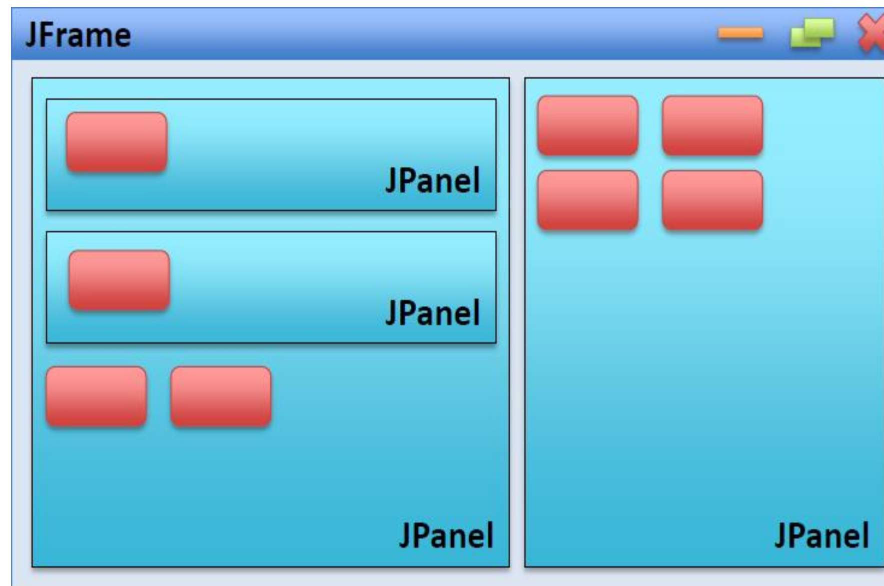
1.4 Graphical User Interface (GUI)

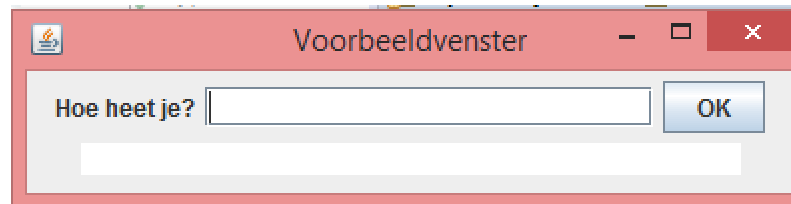
p. 40

GUI

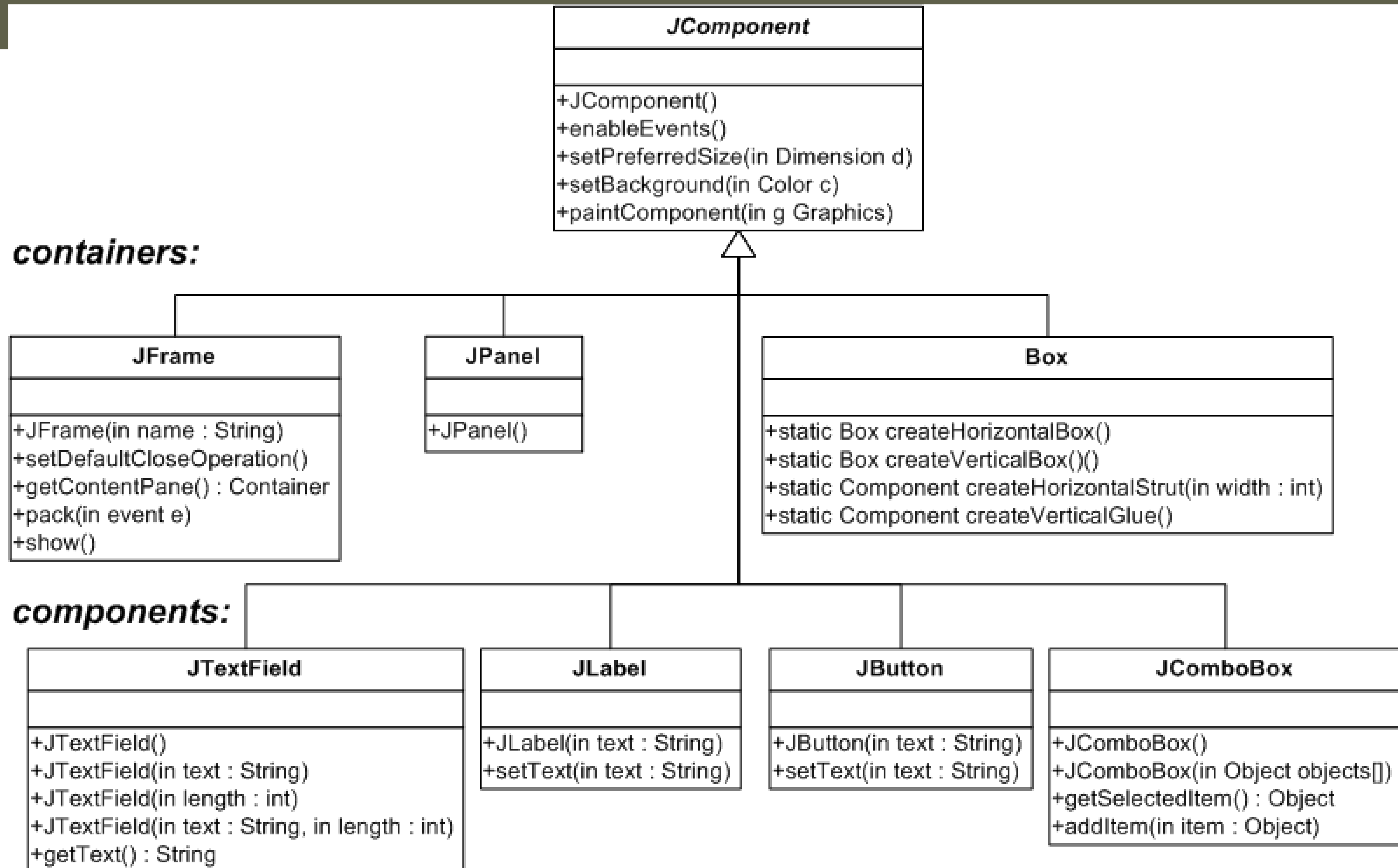
- GUI = Graphical User Interface
- Gebruiken van de javaklassen van Swing
  - Online documentatie
- **Gebaseerd op de *kracht* van object-georiënteerde talen**
  - Encapsulatie
  - Overerving (Inheritance)
  - Abstractie/polymorphisme
    - interfaces

GUI-componenten



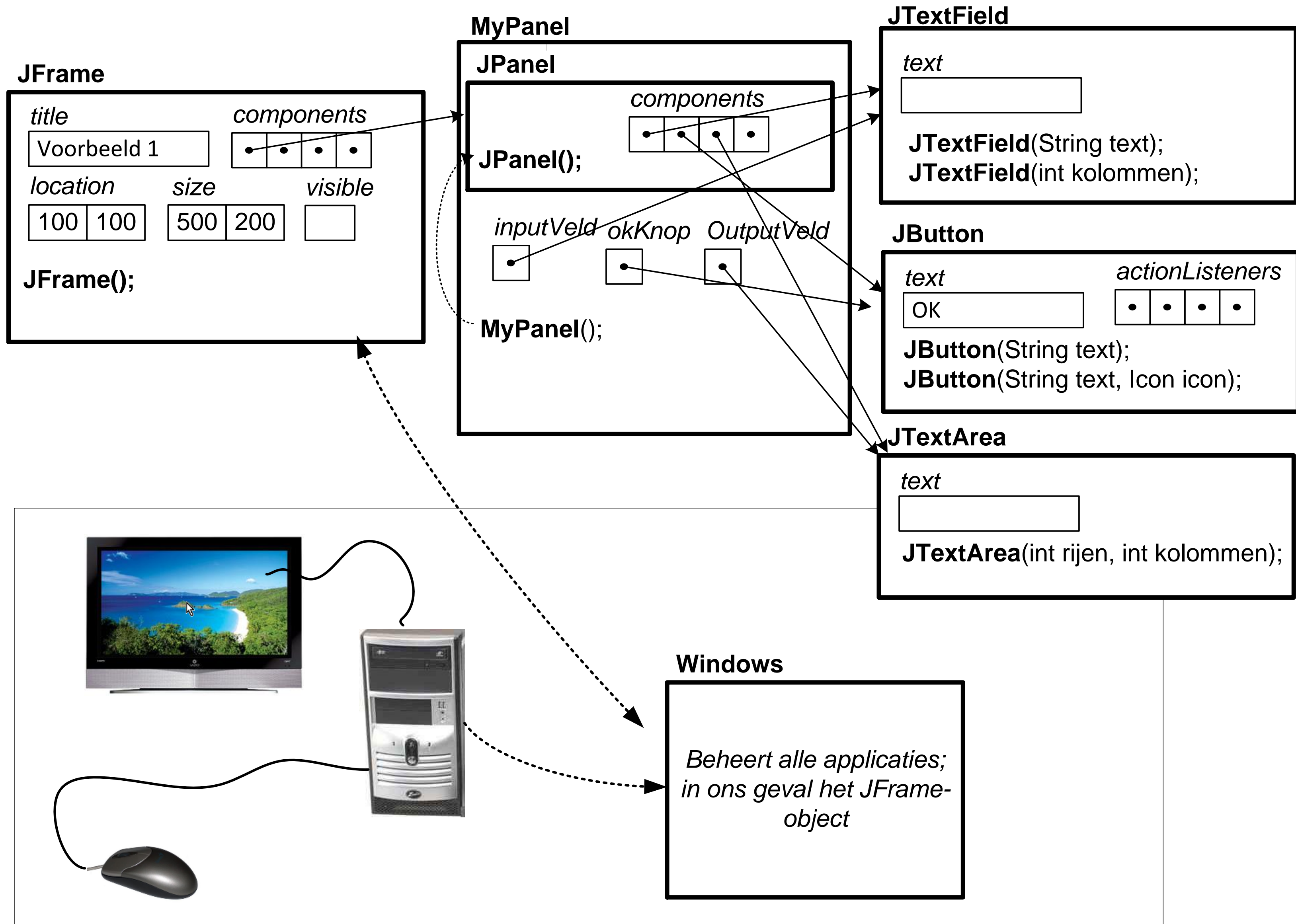


Swing’s klassehiërarchie



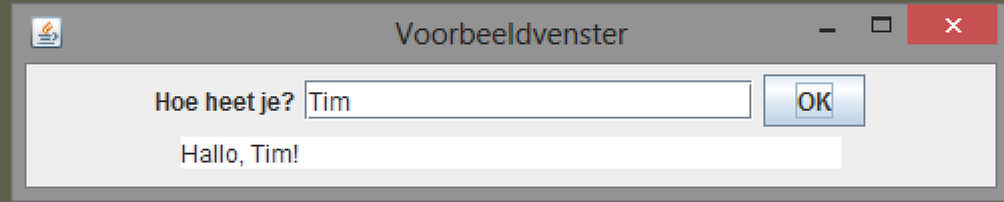
```java
import javax.swing.*;

public class MyPanel1 extends JPanel {

  // GUI-componenten
  private JTextField inputVeld;
  private JButton okKnop;
  private JTextArea outputVeld;

  public MyPanel1()
  {
    inputVeld = new JTextField(20);  // groote veld (# symbolen)
    okKnop = new JButton("OK");  // inhoud knopje
    outputVeld = new JTextArea(1,30); // dimensies veld (rijen,
kolommen)

    // toevoegen van veldjes aan de GUI
    this.add(new JLabel("Hoe heet je?")); // Jlabel: tekst op je GUI
    this.add(inputVeld);
    this.add(okKnop);
    this.add(outputVeld);
  }
```

```
 public static void main(String[] args)
  {
    JFrame frame = new JFrame();
    frame.setSize(400, 100);  // dimensies van het venster
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // zorgt dat
het programma afsluit bij het klikken op de 'X'-knop
    frame.setTitle("Voorbeeldvenster");  // titel van het venster
    frame.setLocation(100, 100); // positie van de linkerbovenhoek van
het venster op het bureaublad

    JPanel hoofdpaneel = new MyPanel1();  // maak MyPanel1-object aan
    frame.add(hoofdpaneel);  // stop het in de Frame

    frame.setVisible(true);  // het venster wordt geactiveerd
  }
}
```



JFrame
title
Voorbeeld 1
components
location
100
100
size
500
200
visible
JFrame();
MyPanel
JPanel
components
JPanel();
inputVeld
okKnop
OutputVeld
MyPanel();
JTextField
text
JTextField(String text);
JTextField(int kolommen);
JButton
text
OK
actionListeners
JButton(String text);
JButton(String text, Icon icon);
JTextArea
text
JTextArea(int rijen, int kolommen);
Windows
Beheert alle applicaties; in ons geval het JFrame-object

Events




```
public class MyPanel2 extends JPanel implements ActionListener {

        // GUI-componenten
        private JTextField inputVeld;
        private JButton okKnop;
        private JTextArea outputVeld;

        public MyPanel2()
        {
                …

                okKnop.addActionListener(this);  // associatie ActionListener

                ...
        }
        // Dit wordt uitgevoerd wanneer er op de knop gedrukt wordt
        public void actionPerformed(ActionEvent e) {
                outputVeld.setText("Hallo, " +  inputVeld.getText() + "!"
        }
}
```

Interface ActionListener

```
interface ActionListener {

    public void actionPerformed(ActionEvent e);

}
```

interface ActionListener

void actionPerformed(ActionEvent e);

ActionEvent

command

>Kopieer>

parameters

ACTION_PERFORMED...

MyPanel

JPanel

components

JPanel();

void actionPerformed(ActionEvent e)
{
…
}

JFrame

title

Voorbeeld 1

components

location

100 100

size

500 200

visible

JFrame();

oproepen actionPerformed() van alle actionListeners met ActionEvent

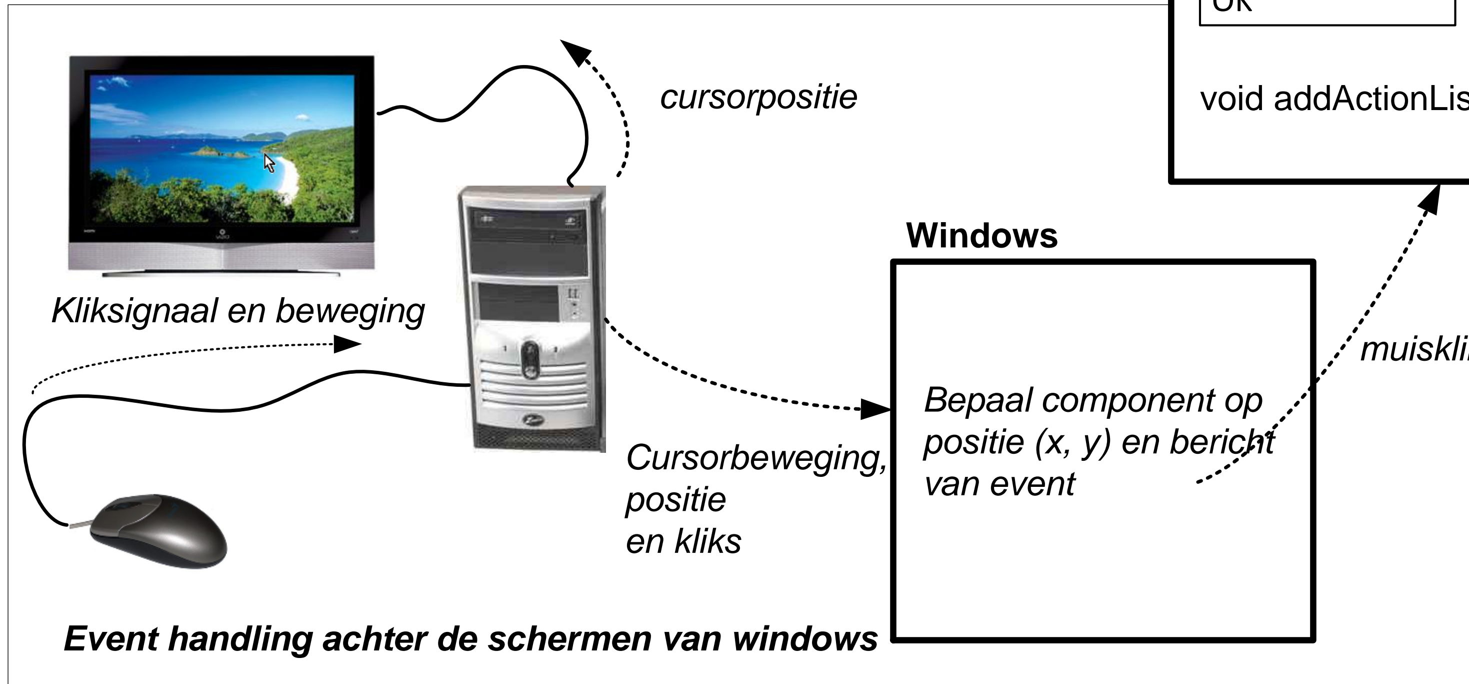
JButton

text

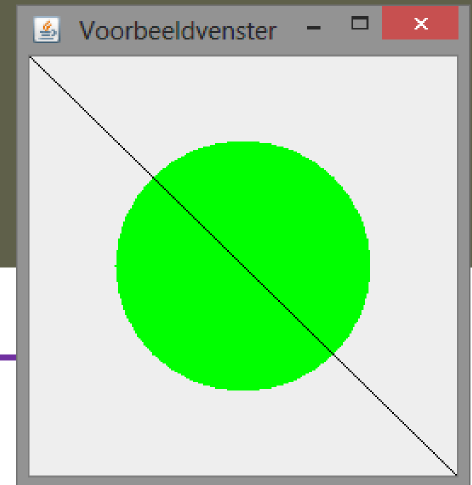
OK

actionListeners

void addActionListener();



Graphics




```
public class MyPanel3 extends JPanel {

	public void paintComponent(Graphics g)
	{
		g.setColor(Color.GREEN);
		g.fillOval(50,50,150,150);
		g.setColor(Color.BLACK);
		g.drawLine(0,0,250,250);
	}

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

MyPanel

JTextField

```
void paintComponent(Graphics g){
  ...
}
```

JFrame

title: Voorbeeld 1

components: • • • •

```
void paintComponent(Graphics g){
  teken frame
  zet titel
  voor alle componenten i:
components[i].paintComponent(g);
}
```

JPanel

components: • • • •

```
void paintComponent(Graphics g){
  teken frame
  zet titel
  voor alle componenten i:
components[i].paintComponent(g);
}
```

JButton

```
void paintComponent(Graphics g){
  ...
}
```

```
void paintComponent(Graphics g){
  super.paintComponent(g);
  g.drawOval(100, 100, 10, 10);
  ...
}
```

Windows

Bepaal welke componenten getekend en hertekend moeten worden

Graphics

Expliciet hertekenen doe je door de repaint() van je frame of panel op te roepen.
Deze geeft dan het commando door aan Windows

Je roept `paintComponent(Graphics g)` dus nooit zelf de op!

Hertekenen doe je door repaint() op te roepen.

Je tekent enkel binnen de `paintComponent(Graphics g)`!

Je mag wel een andere methode oproepen en het Graphics-object doorgeven.

Graphics

Method Summary	
abstract void	clearRect(int x, int y, int width, int height) Clears the specified rectangle by filling it with the background color of the current drawing surface.
void	draw3DRect(int x, int y, int width, int height, boolean raised) Draws a 3-D highlighted outline of the specified rectangle.
abstract void	drawArc(int x, int y, int width, int height, int startAngle, int arcAngle) Draws the outline of a circular or elliptical arc covering the specified rectangle.
void	drawChars(char[] data, int offset, int length, int x, int y) Draws the text given by the specified character array, using this graphics context's current font and color.
abstract boolean	drawImage(Image img, int x, int y, ImageObserver observer) Draws as much of the specified image as is currently available.
abstract void	drawLine(int x1, int y1, int x2, int y2) Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.
abstract void	drawOval(int x, int y, int width, int height) Draws the outline of an oval.
abstract void	drawPolygon(int[] xPoints, int[] yPoints, int nPoints) Draws a closed polygon defined by arrays of x and y coordinates.
abstract void	drawPolyline(int[] xPoints, int[] yPoints, int nPoints) Draws a sequence of connected lines defined by arrays of x and y coordinates.
void	drawRect(int x, int y, int width, int height) Draws the outline of the specified rectangle.
abstract void	drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight) Draws an outlined round-cornered rectangle using this graphics context's current color.

1.4.4. Animaties en Timers

$\Rightarrow$ Oefeningen (WPOs)

Project

Projectonderwerp

- Voldoende programmatorische complexiteit
  - Klassen & objecten
- Spelelement wordt meestal gekozen
  - Creativiteit!
- Mag:
  - Artificiële Intelligentie (A.I.): ‘slim’ algoritme
    - Hoofdstuk 5

Groepjes van 2 - 3

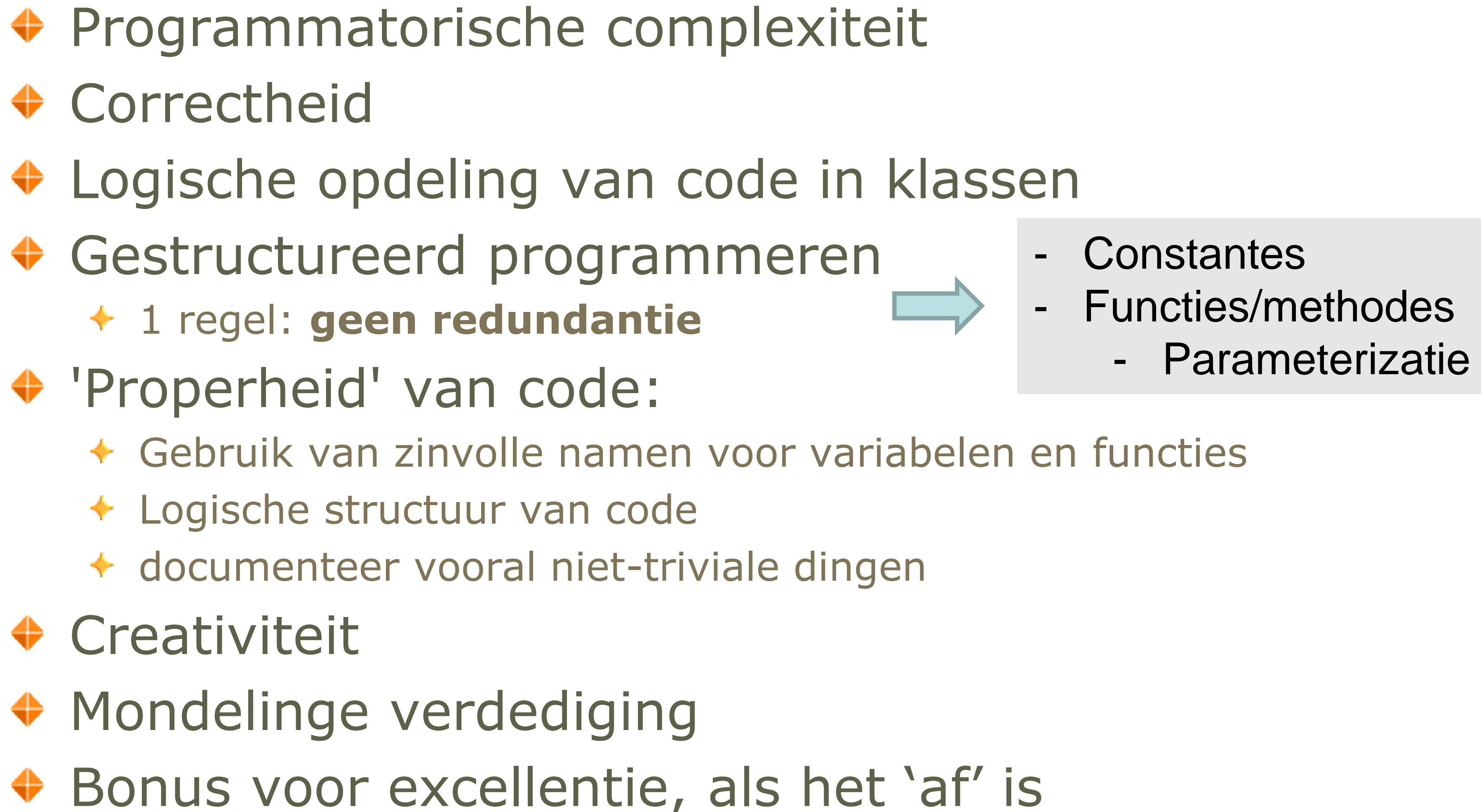
Project: doelstellingen

- Programmeren
- Programmatorische complexiteit, gestructureerd programmeren
- Projectwerk, teamwork
- Creativiteit, ingenieuziteit
- Plezier
- Ontwikkelen probleemoplossende vaardigheden

Project: uitvoering

- Voor paasvakantie: keuze groep & onderwerp

Evaluatie



Laatste tips

- Ingenieur = efficiëntie
- Anderzijds: excellentie vergt net dat tikkeltje extra
- Probleemoplossende vaardigheden (debuggen)
- Als je probleem niet kan oplossen: VRAAG
  - Blijf er niet bij zitten!!
- Referenties en gebruikte code opgeven!
  - **Anders: plagiaat**

Twee voorstellen

- Roblox-game
  - Maak de NPCs slim!
  - Met de taal Lua (lijkt op python)
- State.io is zoals Risk
  - https://play.google.com/store/apps/details?id=io.state.fight
  - Tegenstanders = algoritme = redelijk dom
- Denkspel met Booleaanse functies
  - Zoals Circuit Scramble (niet meer te downloaden…)

https://apkpure.com/nl/circuit-scramble-computer-logic-puzzles/com.Suborbital.CircuitScramble#com.Suborbital.CircuitScramble-2

Deze games wil ik graag mee begeleiden

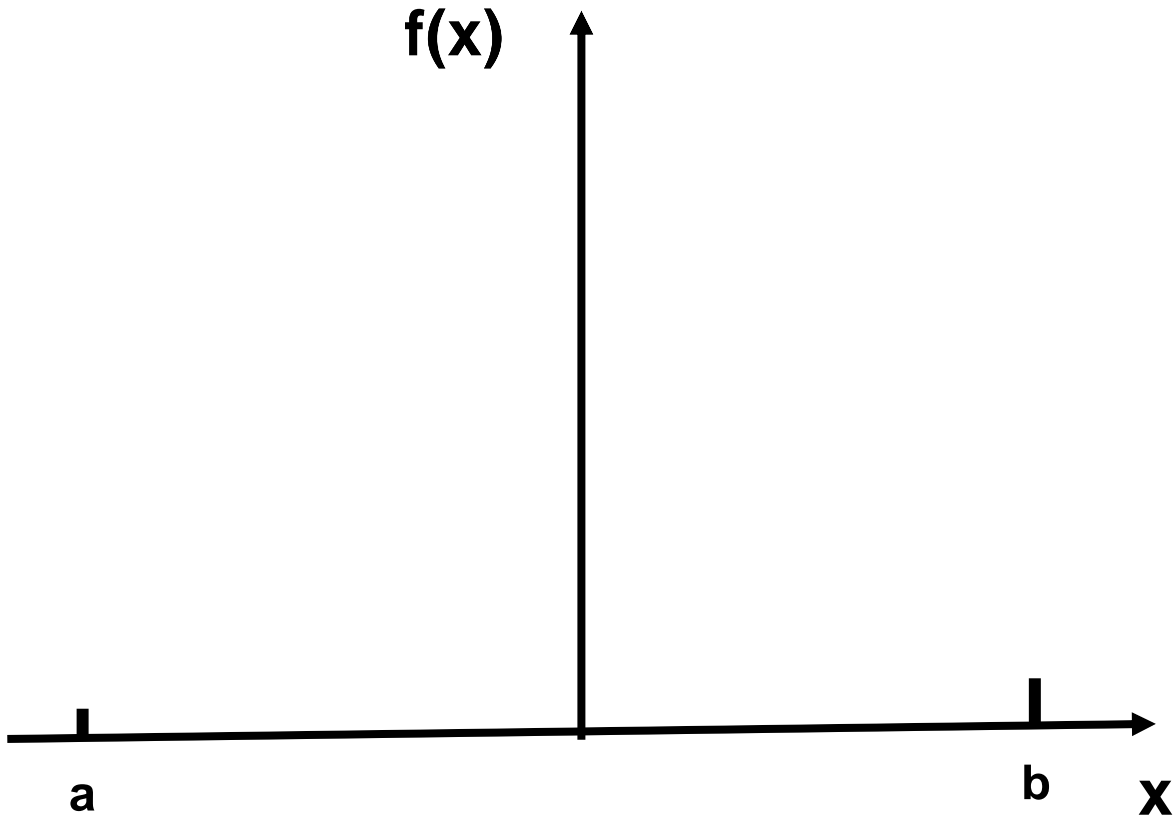
Basisalgoritmen

Hoofdstuk 4

Oefening

Iets met een functie

```java
public class IetsMetFunctie {
	/** PROGRAMMA  */
	public static void main(String[] args) {
		double a = -10, b = 10;

		double xx = vindXX(a, b);
		System.out.println("XX voor deze functie is "+xx);
	}
	public static double f(double x){
//		return 2 *(x - 3);
		return x * x  + 12 *(x - 3);
//		return - x * x * x / 8 + x * x  + 20 *(x - 3);
	}
	public static double vindXX(double a, double b){
		final double PRECISIE = 0.001; // configuratieparameter

		if ( !(f(a) < 0 && f(b) > 0))
			throw new IllegalArgumentException("Geef betere beginpunten!");

		double m = (a + b) / 2;

		while( Math.abs( f(m)) > PRECISIE){
			System.out.print(m+", ");
			if (f(m) < 0)
				a = m;
			else
				b = m;
			m = (a + b) / 2;
		}
		System.out.println(m);
		return m;
	}
```


f(x)
a
b
x

$$F(r)=0.840376\frac{2r+1}{(r+1)^2}-0.221538\frac{8r^2-3r-1}{(r+1)^3}-0.0173983\frac{54r^3-145r^2+12r+3}{(r+1)^4}+0.0214648\times\frac{32r^4-203r^3+161r^2-5r-1}{(r+1)^5}+0.0026529\frac{5+30r-2010r^2+5376r^3-2835r^4+250r^5}{(r+1)^6}$$

Voor welke r-waarden wordt F(r) nul?

- Analytisch niet altijd oplosbaar...
- Voor waarden van r de functie uitrekenen

Abstractie: Functie

Old school

```
/** PROGRAMMA  */
public static void main(String[] args) {
    double a = -10, b = 10;

    double nulpunt = vindNulpunt(a, b, 1);
    System.out.println("XX voor deze functie is "+nulpunt);
}
public static double f(double x, int functie){
    switch (functie) {
    case(1): return 2 *(x - 3);
    case(2): return x * x  + 12 *(x - 3);
    case(3): return - x * x * x / 8 + x * x  + 20 *(x - 3);
    default: throw new IllegalArgumentException("Functie "+functie+" is geen geldige parameter.");
    }
}
public static double vindNulpunt(double a, double b, int functie){
    final double PRECISIE = 0.000001; // configuratieparameter

    if ( !(f(a, functie) < 0 && f(b, functie) > 0))
        throw new IllegalArgumentException("Geef betere beginpunten!");

    double m = (a + b) / 2;
    int ctr=0;
    while( Math.abs( f(m, functie)) > PRECISIE){
        System.out.print(m+", ");
        if (f(m, functie) < 0)
            a = m;
        else
            b = m;
        m = (a + b) / 2;
```

Functie als object

Functie1

```
public double f(double x)
    return 2 *(x - 3);
```

```
public static double vindNulpunt(double a, double b, Functie functie){
    if ( !(functie.f(a) < 0 && functie.f(b) > 0))
        throw new IllegalArgumentException("Geef betere beginpunten!");

    double m = (a + b) / 2;

    while( Math.abs( functie.f(m)) > 0.01){
        System.out.print(m+", ");
        if (functie.f(m) < 0)
            a = m;
        else
            b = m;
        m = (a + b) / 2;
    }
    System.out.println(m);
    return m;
}
```

```
public interface Functie {
/** geeft functiewaarde in
opgegeven punt */
    double f(double x);
}
```

Interfaces

7. Een **abstracte methode** heeft *geen implementatie* (deel tussen accolades), enkel een header [`abstract`]. De eerste lijn eindigt met een punt-komma.
8. Een **interface** is een klasse met *enkel abstracte methodes* [`interface`].
   - Een interface heeft geen constructor en geen attributen (al zijn statische attributen wel mogelijk).
   - Een klasse mag meerdere interfaces als superklasse hebben [`implements`]: "de klasse implementeert de interface".
   - Een concrete klasse moet alle abstracte methodes implementeren. Anders blijft het een abstracte klasse en kan je er geen objecten van maken.

Voordeel van abstractie

Librarycode (niet aanpasbaar)

interface Functie

```
double f(double x);
```

```
double vindNulpunt(a, b, Functie functie)
{
   ...
```
Code gebruik makend van `functie` en enkel de methoden van interface `Functie`
```
   ...
}
```

Mijn code

class Functie1

```
public double f(double x) {
        return 2 *(x - 3);
}
```

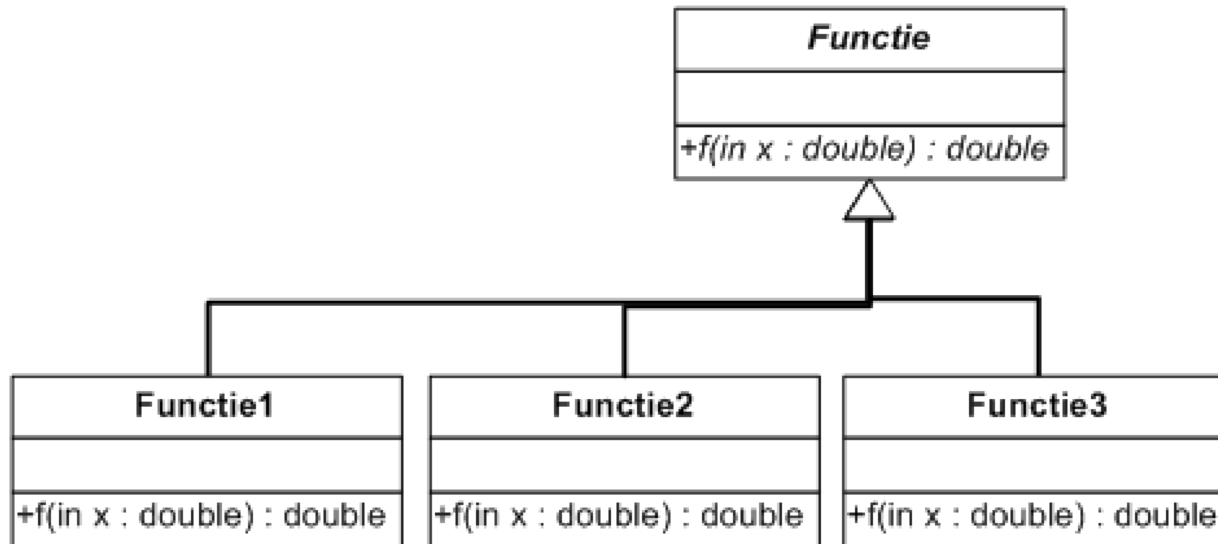
```
public static void main(String[] args)
{
 Functie functie = new Functie1();

 double x = vindNulpunt(a, b, functie);
}
```

Functie: iets abstracts

- Nulpunt zoeken via een Functie object
- *De methode werkt immers voor alle functies*
- Definiëren wat een *functie* is:
  - Methode: double f(double x)
- Hiërarchie van functies

```java
public interface Functie {
/** geeft functiewaarde in opgegeven punt */
    double f(double x);
}
class Functie1 implements Functie{
   public double f(double x) {
        return 2 *(x - 3);
   }
}
class Functie2 implements Functie{
   public double f(double x) {
        return x * x  + 12 *(x - 3);
   }
}
class Functie3 implements Functie{
   public double f(double x) {
   return - x * x * x / 8 + x * x  + 20 *(x - 3);
   }
}
```

Klasses

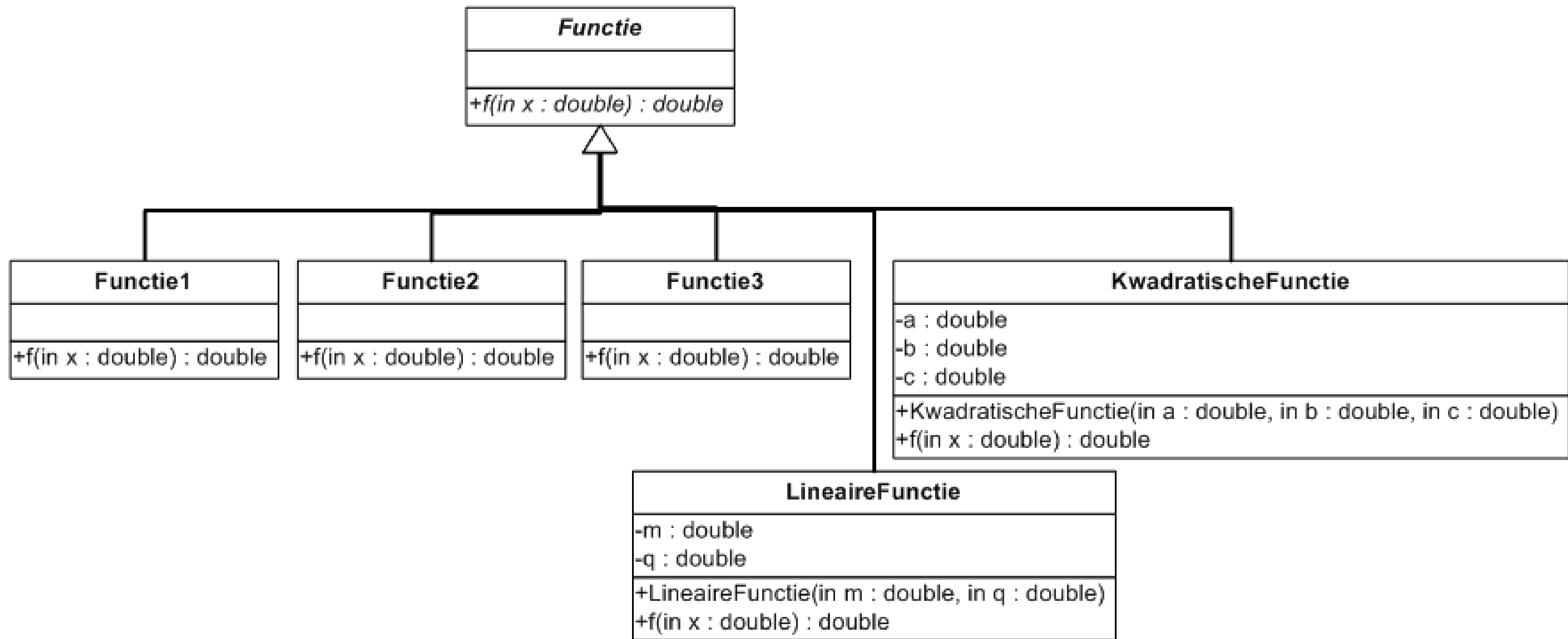


Gebruik

- Constructor: hier zonder parameters
- Default constructor: moet je niet definiëren
  - Als je een andere constructor definieert, vervalt deze. Met de andere constructor geef je aan dat je parameters moet meegeven.

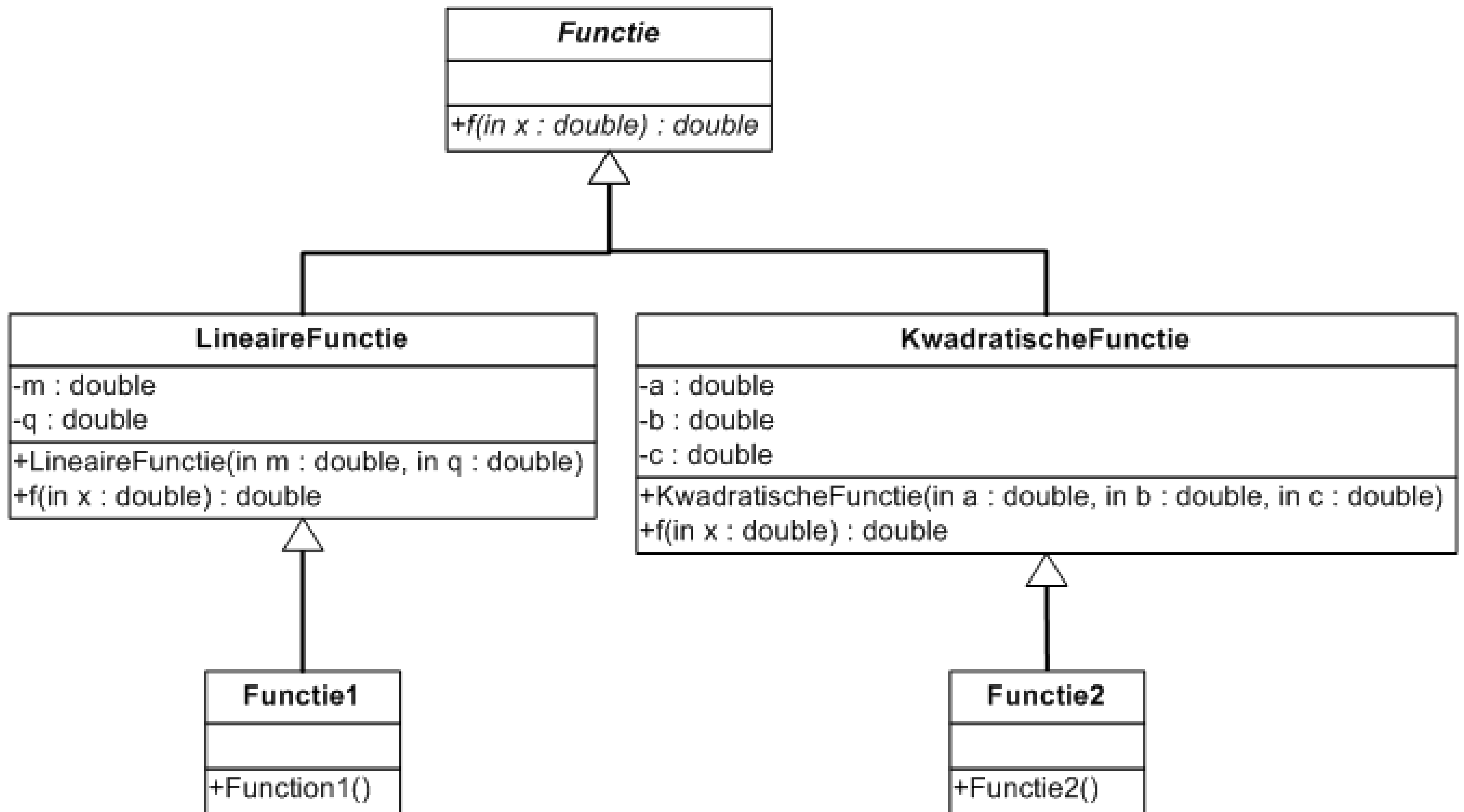
```
Functie functie = new Functie2();
double nulpunt = vindNulpunt(a, b, functie);
```

Extra klasses



```
class LineaireFunctie implements Functie{
  double m, q;
  public LineaireFunctie(double m, double q){
       this.m = m;
       this.q = q;
  }
  public double f(double x) {
       return m * x + q;
  }
}
class KwadratischeFunctie implements Functie{
  double a, b, c;
  public KwadratischeFunctie(double a, double b, double c){
       this.a = a;
       this.b = b;
       this.c = c;
  }
  public double f(double x) {
       return a * x * x + b * x + c;
  }
}
```

```
Functie functie = new KwadratischeFunctie(1, 12, -36);
double nulpunt = vindNulpunt(a, b, functie);
```


Functie
+f(in x : double) : double
LineaireFunctie
-m : double
-q : double
+LineaireFunctie(in m : double, in q : double)
+f(in x : double) : double
KwadratischeFunctie
-a : double
-b : double
-c : double
+KwadratischeFunctie(in a : double, in b : double, in c : double)
+f(in x : double) : double
Functie1
+Function1()
Functie2
+Functie2()

Functie met specifieke parameters

```
class LineaireFunctie implements Functie{
  double m, q;
  public LineaireFunctie(double m, double q){
        this.m = m;
        this.q = q;
  }
  public double f(double x) {
        return m * x + q;
  }
}
```

```
class Functie1 extends LineaireFunctie{
        Functie1(){
                super(2, -6);
        }
}
```

```
class KwadratischeFunctie implements Functie{
  double a, b, c;
  public KwadratischeFunctie(double a, double b, double c){
       this.a = a;
       this.b = b;
       this.c = c;
  }
  public double f(double x) {
       return a * x * x + b * x + c;
  }
}
```

```
class Functie2 extends KwadratischeFunctie{
  Functie2(){
       super(1, 12, -36);
  }
}
```

Lambda-expressies

TER INFO
NIET TE KENNEN

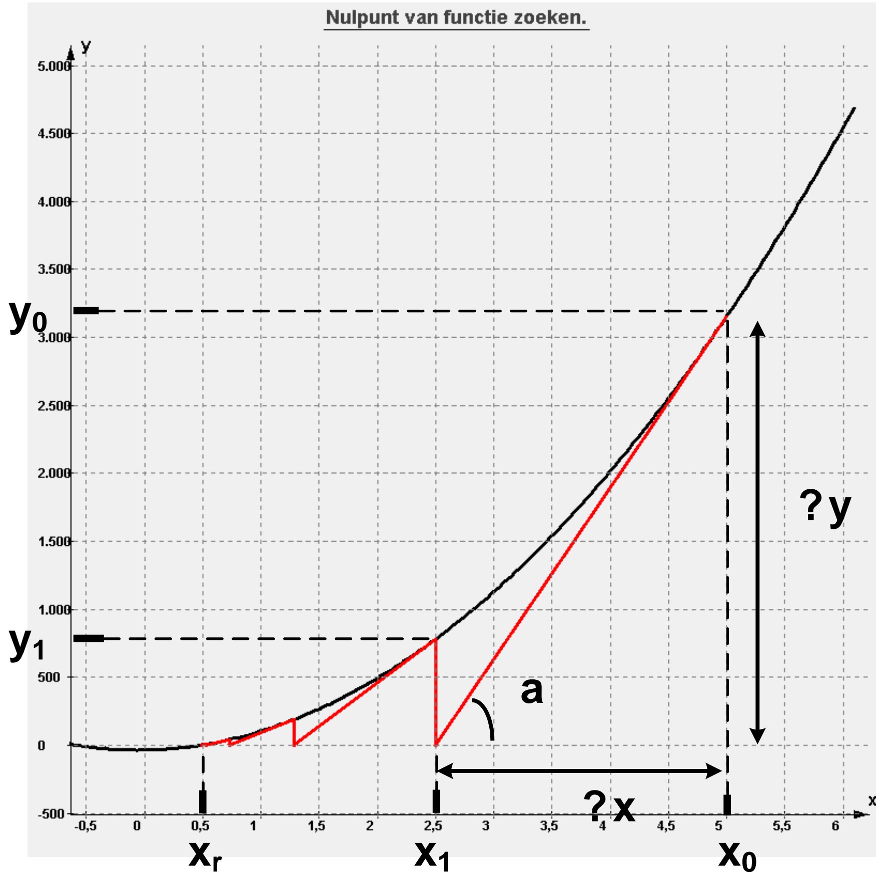
- Sinds Java 8 (versienummer 1.8) kan je functies op een eenvoudigere manier meegeven, namelijk als **lambda-expressies**.

```
double nulpunt = vindNulpunt(a, b, x -> 2*x + x - 3);
```

- Je definieert ze met een pijl '**->**': voor de pijl zet je de inputparameters (hier: x), na de pijl zet je de berekening van de output. De Javacompiler checkt dat de gegeven lambda-expressie voldoet aan de gevraagde interface *Functie*, wat hier het geval is!

Newton's algoritme

Nulpunten zoeken: Newton's algoritme



- (x_0, y_0): eerste gok

$$\frac{\Delta y}{\Delta x} = \frac{\partial f}{\partial x} \Leftrightarrow \Delta x = {}^{\Delta y}\!\Big/_{\frac{\partial f}{\partial x}} \Leftrightarrow x_1 = x_0 - {}^{\Delta y}\!\Big/_{\frac{\partial f}{\partial x}}$$

Afgeleide van functie kennen

```
interface Functie {
	/** geeft functiewaarde in opgegeven punt */
	double f(double x);
}
```

```
interface FunctieMetAfgeleide extends Functie {
	/** geeft waarde van de afgeleide in opgegeven punt */
	double afgeleide(double x);
}
```

```java
static int nbrIteraties=0;
public static double vindNulpuntMetNewton(FunctieMetAfgeleide functie,
int eersteGok){
        final int MAX_ITERATIES = 150;
        final double PRECISIE = 0.001;

        double nulpunt = eersteGok;
        double fx = functie.f(nulpunt);
        nbrIteraties=0;
        while (Math.abs(fx) > PRECISIE && nbrIteraties < MAX_ITERATIES){
                double dfx = functie.afgeleide(nulpunt);
                if (dfx == 0)
                        throw new RuntimeException("Nulpunt met Newton:
Afgeleide in "+nulpunt+" is nul waardoor Newton faalt.");
                System.out.println("["+nbrIteraties+"] Current =
"+nulpunt+" fx="+fx+" dfx="+dfx+" => new = "+(nulpunt - fx/dfx));
                nulpunt = nulpunt - fx/dfx;
                fx = functie.f(nulpunt);
                nbrIteraties++;
        }
        if (nbrIteraties >= MAX_ITERATIES)
            throw new RuntimeException("Nulpunt met Newton: convergeert
niet, te veel iteraties, beste punt tot nu toe heeft waarde
"+functie.f(nulpunt)+"!");
        System.out.println("Newton nulpunt = "+nulpunt+" met fx="+fx);
        return nulpunt;
}
```

Mogelijke problemen

1. **Geen of trage convergentie**
   - Gradiënt geeft niet noodzakelijk de juiste richting aan
2. **Crash**
   - Afgeleide die nul is
3. **Verticale afgeleide [Ruben Simons 2019]**
   - X-waarde blijft dan dezelfde, we blijven ‘hangen in hetzelfde punt’
4. **Geen nulpunt voor functie**
   - Vraag: *kan het algoritme garanderen dat functie geen nulpunten heeft?*

Discrete Simulaties

Discrete simulatie

- We benaderen de continuë wereld, afgeleiden en integralen benaderen we met Δ's
- Wereld veranderen we elke Δt

$$
\begin{aligned}
\Delta v_x &= a_x . \Delta t \\
\Delta v_y &= a_y . \Delta t \\
\Delta x \quad &= v_x . \Delta t \\
\Delta y \quad &= v_y . \Delta t
\end{aligned}
$$

p. 27

```java
void simulationLoop(double kogelgewicht, double hoek, double vuurkracht){
	final int TIME_STEP = 1;
	final double G = 9.81, FRICTION=0.0001, UNCERTAINTY=0.1;
	Random rand = new Random();

	Kogel kogel = new Kogel(kogelgewicht);
	kogel.vx = vuurkracht  * Math.cos(hoek) / kogelgewicht;
	kogel.vy = vuurkracht  * Math.sin(hoek) / kogelgewicht;

	int time = 1;
	boolean ended = false;

	while(!ended){
		// forward time
		time += TIME_STEP;

		// update velocity
		double friction = FRICTION * kogel.vy * kogel.vy;
		kogel.vy += - TIMESTEP * G + (kogel.vy < 0 ? friction : -
friction) + (rand.nextDouble() - 0.5) * UNCERTAINTY;

		friction = FRICTION * kogel.vx * kogel.vx;
		kogel.vx += (kogel.vx < 0 ? friction : -friction) +
(rand.nextDouble() - 0.5) * UNCERTAINTY;

		// update position
		kogel.x += kogel.vx * TIMESTEP;
		kogel.y += kogel.vy * TIMESTEP;

		// check collisions
		if (kogel.y < 0)
			ended = true;
```

```java
class Kogel{
	double massa;
	double x, y, vx, vy;
	Kogel(double massa){
		this.massa = massa;
	}
}
```

Uitbreiden

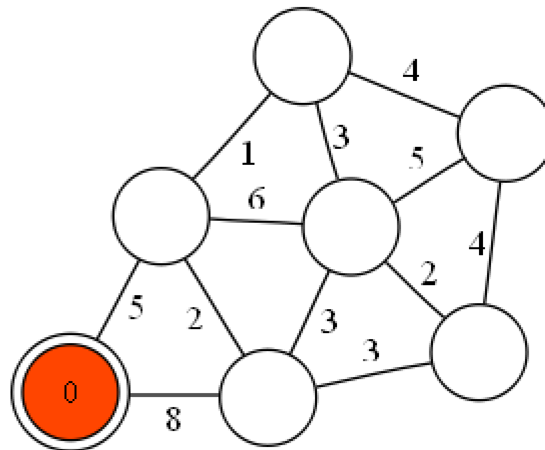
- Alle fysische dingen: MovingObject klasses
- Met subklassen voor specifiek gedrag

Dijkstra's kortste pad

p. 29

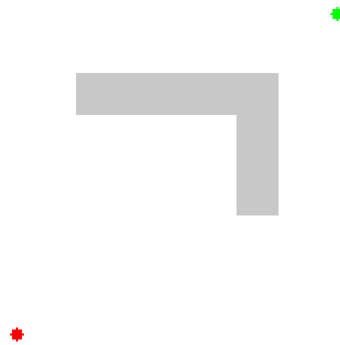
Probleemstelling

- ***Gegeven***: graph waarbij elke link een gewicht heeft

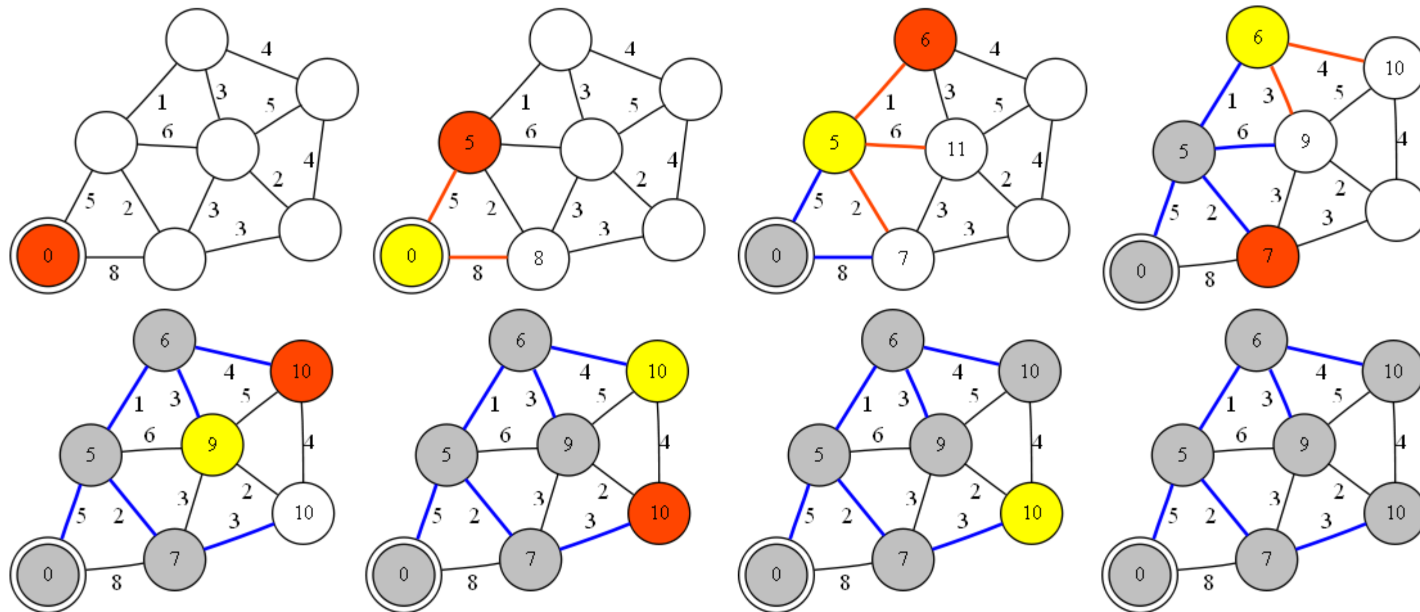


- ***Gevraagd***: kortste pad (minimaliseren som van de gewichten) van elke node naar node 0

Toegepast op een labyrinth



Dijkstra’s algoritme



Het algoritme maakt een priority queue van nodes aan, die gesorteerd wordt op afstand (kleinste eerst).
Het algoritme onthoudt per node:

- de tot-dan-toe gevonden afstand naar node 0. Initieel op oneindig gezet.
- de eerstvolgende node die hij moet nemen om naar node 0 te gaan (in blauw aangegeven in graaf).

Afstand van node 0 zet je op 0. De node voeg je toe aan de priority queue.
Doe zolang priority queue niet leeg is:

- Neem eerste element uit priority queue (kortste afstand tot node 0)
- Ga voor deze node al zijn buren af:
    - Tel de afstand tot buur bij je eigen afstand.
    - Kijk of deze afstand beter is dan de tot-dan-toe gevonden afstand van de buur. Indien dit zo is, update de tot-dan-toe gevonden afstand en geef aan dat de buur naar de node moet gaan.
    - Als de buur nog niet in de priority queue is, voeg hem toe.

Slimme algoritmen

Deel I

Hoofdstuk 5

Beslissingsalgoritme

- Doel bereiken
- Aantal acties ter beschikking
- ⇒ Actiesequentie bedenken om doel te bereiken

p. 31

Indeling volgens oplossingsmethode

- **Type 1**: De oplossing kan berekend worden met een formule (analytisch).
- **Type 2**: Je kunt de oplossing gericht zoeken of construeren (rechttoe-rechtaan).
- **Type 3**: Je gaat alle mogelijke actiesequenties af om een oplossing te vinden.
- **Type 4**: Door slimme keuzes (*heuristieken*) te maken, kan je verschillende actiesequenties uitsluiten.
- **Type 5**: Je leert al doende welke de juiste keuzes zijn.

Type 1: analytisch

1. Je kan oplossing berekenen met formule
   - *Analytisch*
   - *Voorbeeld:* nulpunten van kwadratische vergelijking

Type 2: gerichte ‘constructie’

Je kan oplossing gericht zoeken met quasi-zekerheid om te arriveren

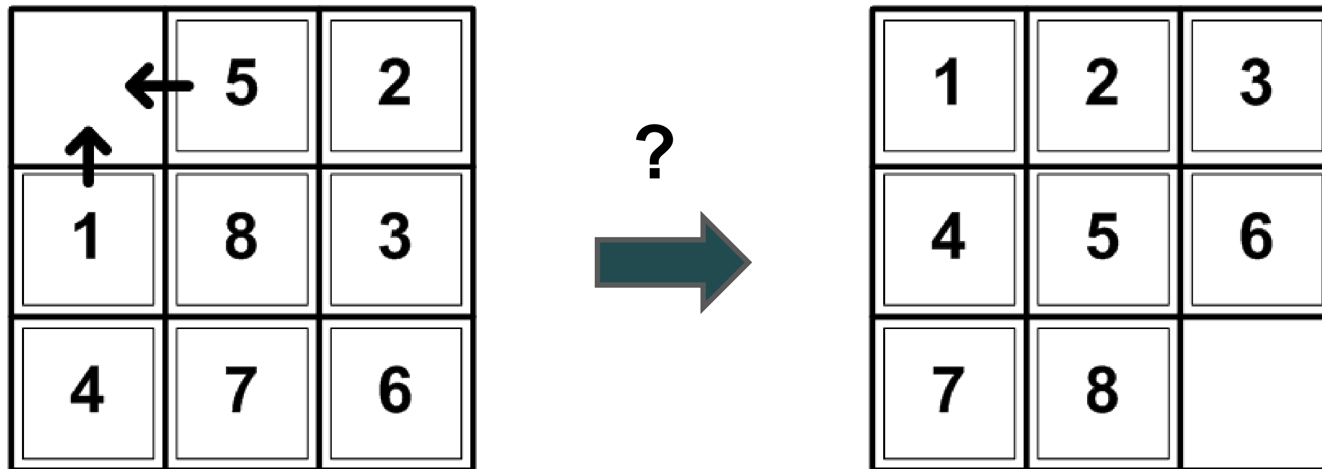
Voorbeelden:

- Nulpunt van een functie zoeken met benaderingsalgoritme
- priemgetallen met zeef van Erathosteness
- Berekening grootst-gemene deler
- Dijkstra

p. 31

De schuifpuzzel

Puzzel 1 in schuifpuzzel.jar



- Staat van puzzel: posities van stukjes
- Mogelijke acties: LEFT, UP, RIGHT, DOWN
  - Niet alle 4 steeds mogelijk
- **We gaan 6 algoritmen zien**

Java app

We hebben een app gemaakt waarmee je de 6 algoritmen kan vergelijken en zelf ook het spel spelen. Te vinden als **SchuifPuzzel.jar** op parallel website (Theorie – Les 7) of als **Schuifpuzzel.java** in package **Schuifpuzzel**.

Deze code verschilt van die uit **package zoeken** om het spel interactief te maken.

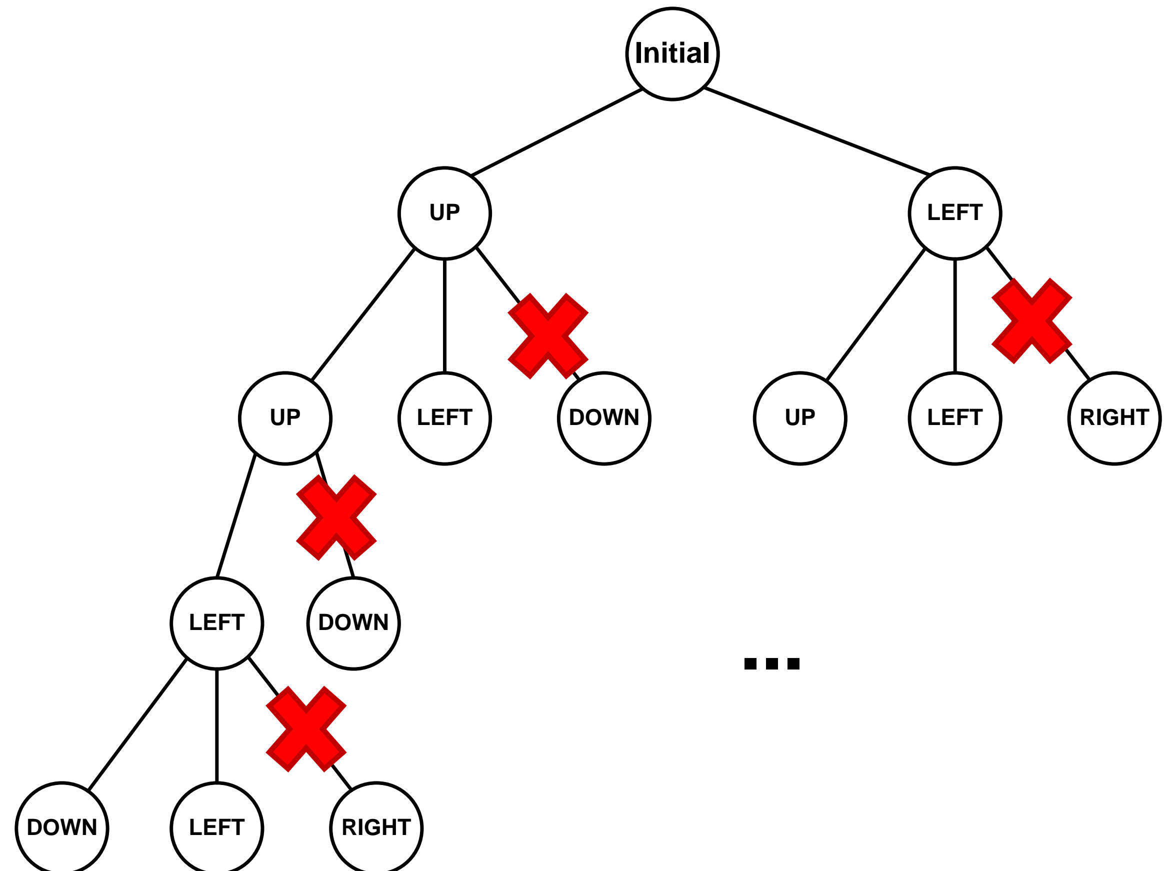
We tellen het aantal **bekeken** nodes (niet bezochte nodes) als maat voor de zoektijd.

Type 3: Alle mogelijkheden afgaan

- Je weet niet in welke richting de oplossing ligt.
  - Voor sommige problemen is men er vrijwel zeker van dat je alle mogelijkheden af moet lopen (*NP-complete problemen, zie later*)
  - Je genereert alle mogelijke sequenties van akties en kijkt welke tot *een* oplossing of tot *de beste* oplossing leidt.
- **het scannen van de volledige zoekruimte is nodig…**

Zoekruimte = zoekboom

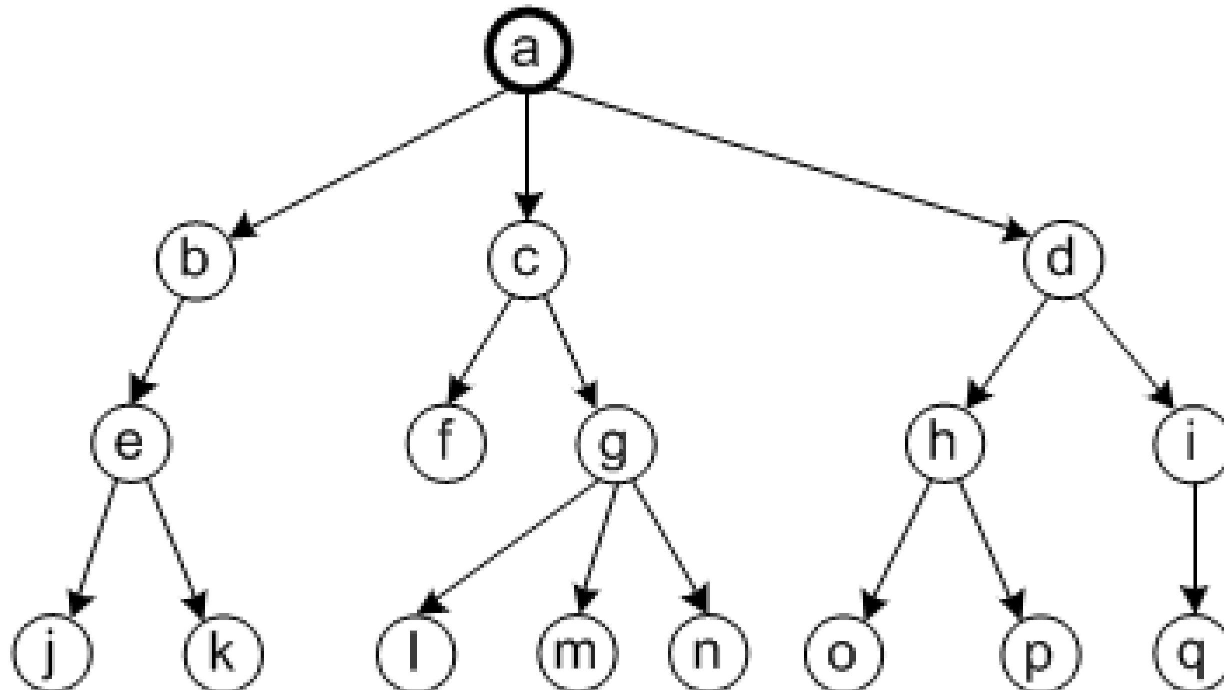
	5	2
1	8	3
4	7	6



Doorploeteren zoekboom

= Brute-force search

- 2 mogelijkheden:
  - **depth-first : a-b-e-j-k-c-f-g-l-m-n-d-h-o-p-i-q**
    - = ***backtracking***
  - **breadth-first: a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q**



Sudoku met backtracking

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

- http://en.wikipedia.org/wiki/Backtracking

(1) Backtracking

```java
public interface ZoekNode<Zet> {
    List<Zet> possibleMoves();
    void move(Zet zet);
    void undoMove(Zet zet);
    boolean isSolution();
    ZoekNode<Zet> clone();
}
```

```java
/** backtracking
 * toepassing: sudoku
 */
public static <Zet> boolean backtracking (ZoekNode<Zet> node){
        List<Zet> zetten = node.possibleMoves();

        for(Zet zet: zetten ){
                node.move(zet);
                if (node.isSolution()){
                        System.out.println("Oplossing gevonden: "+node+"!");
                        return true;
                }
                if (backtracking(node))
                        return true; // stop met verder zoeken

                node.undoMove(zet); // keer terug
        }
        return false; // geen children of geen enkele leidde naar oplossing
}
```

Generieke implementatie

- Via abstractie is het algoritme *algemeen bruikbaar,* voor elk probleem!
  - Enkel interface implementeren
- Code + uitleg staat op website
  - Je moet enkel de code van de algoritmen begrijpen, niet de implementatie van ZoekNode

```
public interface ZoekNode<Zet> {
        List<Zet> possibleMoves();
        void move(Zet zet);
        void undoMove(Zet zet);
        boolean isSolution();
        ZoekNode<Zet> clone();
}
```

Backtracking met gelimiteerde diepte

```
static int MAXIMALE_DIEPTE = 10;

public static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte
(ZoekNode<Zet> node, int max_diepte)
{
        MAXIMALE_DIEPTE = max_diepte;
        return backtrackingMetPadEnMaximaleDiepte_rec(node, 1);
 }
```

```java
private static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte_rec
(ZoekNode<Zet> node, int diepte){
  List<Zet> zetten = node.possibleMoves();

  for(Zet zet: zetten ){
     node.move(zet);
     if (node.isSolution()){
     // pad wordt aangemaakt bij het terugkeren uit de recursie
          List<Zet> pad = new ArrayList<Zet>();
          pad.add(zet);
         return pad;
     }
     if (diepte < MAXIMALE_DIEPTE){
        List<Zet> pad = backtrackingMetPadEnMaximaleDiepte_rec(node,
diepte+1);
        if (pad != null){
           pad.add(0, zet); // voeg vooraan toe
           return pad; // stop met verder zoeken
        }
     }
     node.undoMove(zet); // keer terug
   }
   return null; // geen children of geen enkele leidde naar oplossing
 }
```

```java
private static <Zet> List<Zet> backtrackingMetPadEnMaximaleDiepte_rec
(ZoekNode<Zet> node, int diepte){
  List<Zet> zetten = node.possibleMoves();

  for(Zet zet: zetten ){
     node.move(zet);
     if (node.isSolution()){
     // pad wordt aangemaakt bij het terugkeren uit de recursie
         List<Zet> pad = new ArrayList<Zet>();
         pad.add(zet);
        return pad;
     }
     if (diepte < MAXIMALE_DIEPTE){
        List<Zet> pad = backtrackingMetPadEnMaximaleDiepte_rec(node,
diepte+1);
        if (pad != null){
           pad.add(0, zet); // voeg vooraan toe
           return pad; // stop met verder zoeken
        }
     }
     node.undoMove(zet); // keer terug
   }
   return null; // geen children of geen enkele leidde naar oplossing
 }
```

Code breadth-first

```
public static <Zet> boolean breadthfirst (ZoekNode<Zet> startnode){
   FIFOQueue<ZoekNode<Zet>> openNodes = new FIFOQueue<ZoekNode<Zet>>(1000);
   openNodes.add(startnode);

   while(!openNodes.isEmpty()){
        ZoekNode<Zet> node = openNodes.get();

        List<Zet> zetten = node.possibleMoves();
        for(Zet zet: zetten ){
               ZoekNode<Zet> child = node.clone(); //copy!!
               child.move(zet);
               if (child.isSolution()){
                      System.out.println("Oplossing gevonden!");
                      return true;
               }
               openNodes.add(child);
        }
   }
   return false; // geen oplossing gevonden
 }
```

Conclusies

- (1) Backtracking:
  - zoekboom wordt niet in het geheugen gehouden
  - De recursie houdt het pad bij
    - Pad construeer je bij het terugkeren van de recursie
  - Niet steeds snelste/kortste oplossing
- (2) Breadth-first
  - Je maakt boom aan in het geheugen
    - Steeds kopieen van de nodes nodig
    - Child moet zijn parent bijhouden (is nu niet in code)
    - En eventueel een parent al zijn children
  - Pad haal je uit de boom

‘Domme rekenkracht’

- Brute force: puur rekenen
- Niet echt intelligent
- Echter: zoekruimte wordt snel te groot!
  - 10 stappen > $2^{10} \approx 1000$ mogelijke sequenties
  - 6x6 puzzel > 100 stappen > $2^{100} \approx 10^{30}$ mogelijke sequenties
- Kunnen we gerichter zoeken?

p. 31

Oplossingsmethoden

- **Type 1**: De oplossing kan berekend worden met een formule (analytisch).
- **Type 2**: Je kunt de oplossing gericht zoeken of construeren (rechttoe-rechtaan).
- **Type 3**: Je gaat alle mogelijke actiesequenties af om een oplossing te vinden.
- **Type 4**: Door slimme keuzes (*heuristieken*) te maken, kan je verschillende actiesequenties uitsluiten.
- **Type 5**: Je leert al doende welke de juiste keuzes zijn.

p. 36

Gericht zoeken: (3) *greedy search*

- Kies actie die oplossing korter bij brengt
- Gebaseerd op een ***score***
- Puzzel: *totale afstand tot eindpositie*
  - Manhattan-afstand van stukje tot eindpositie

$$|x_1 - x_2| + |y_1 - y_2|$$

  - Sommeren over alle stukjes => score

8

	5	2
1	8	3
4	7	6

7

1	5	2
	8	3
4	7	6

6

1	5	2
4	8	3
	7	6

5

1	5	2
4	8	3
7		6

4

1	5	2
4		3
7	8	6

3

1		2
4	5	3
7	8	6

2

1	2	
4	5	3
7	8	6

1

1	2	3
4	5	
7	8	6

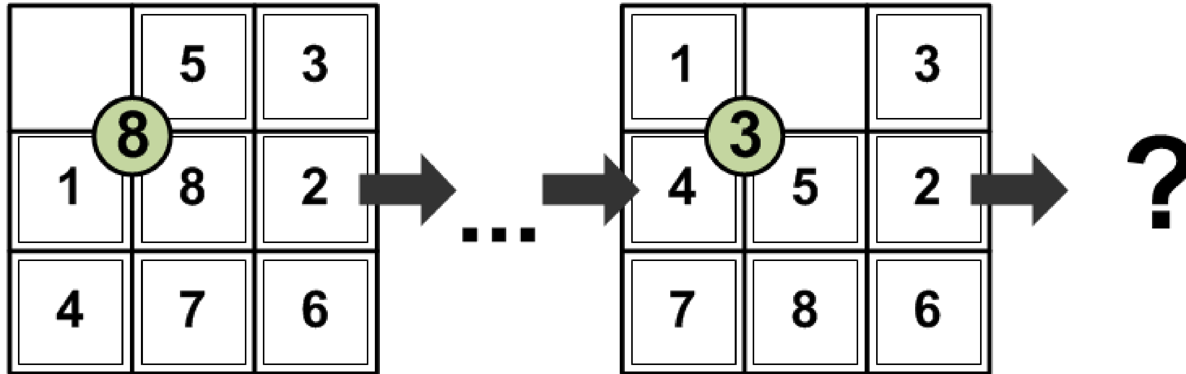
0

1	2	3
4	5	6
7	8	

OK

stu

Puzzel 9 in schuifpuzzel.jar is wel een goede puzzel



Opmerking:

Met app kom je hierop uit, omdat je bij zet 5 twee mogelijkheden hebt

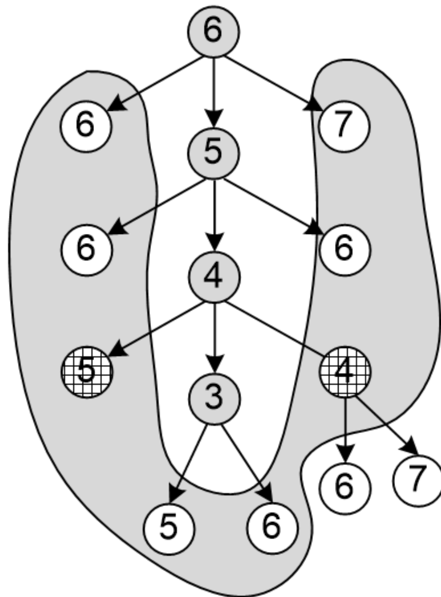
1	5	3
4	2	6
7	8	Score = 2

- terugkeren op stappen (algoritmen 4 & 5)
- of in de toekomst kijken noodzakelijk! Algoritme 6.

(3) Greedy search

Zoekboom p.38 in cursus en schuifpuzzel.jar

Fictieve zoekboom, niet gelinkt aan een schuifpuzzel



4 **bezochte** noden (grijze nodes)

4+8 **bekeken** nodes (grijze nodes + grijze wolk)

https://en.wikipedia.org/wiki/Best-first_search

(4) Best-first search

- Idee student Ruben Pauwels (examen juni 2016): *greedy search* combineren met *backtracking*: als je vast zit, keer je terug op je stappen en neemt het tweede-beste pad.
- Best-first is gebaseerd op dit idee, *maar gebruikt niet letterlijk backtracking*, wel een queue. Veelgemaakte fout op examen!
- Dus: sorteren op score, in een lijst de open nodes bijhouden
  - **Open nodes** = tot waar je gekomen bent in de verschillende takken die je al bekeken hebt
  - **PriorityQueue** gebruiken: sorteert de elementen volgens score (ascending/opklimmend: laagste eerste)
- Implementatie zelfde als breadthfirst, maar dan **PriorityQueue** ipv **FIFOQueue**

Best-first ≠ Greedy + Backtracking!!

Code breadth-first => best-first

```
public static <Zet> boolean bestFirst (ZoekNode<Zet> startnode){
PriorityQueue <ZoekNode<Zet>> openNodes =
                    new PriorityQueue ekNode<Zet>>(1000);
   openNodes.add(startnode);

   while(!openNodes.isEmpty()){
       ZoekNode<Zet> node = openNodes. poll()

       List<Zet> zetten = node.possibleMoves();
       for(Zet zet: zetten ){
              ZoekNode<Zet> child = node.clone(); //copy!!
              child.move(zet);
              if (child.isSolution()){
                     System.out.println("Oplossing gevonden!");
                     return true;
              }
              openNodes.add(child);
       }
   }
   return false; // geen oplossing gevonden
 }
```

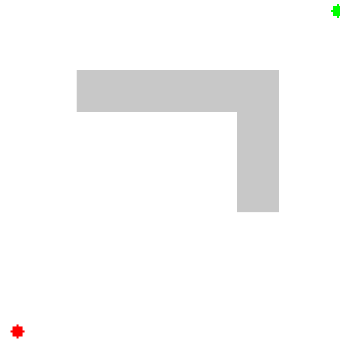
Voorbeeld best-first

Zoekboom p.38 in cursus en schuifpuzzel.jar

We gaan verder waar we gestopt zijn met greedy search

Als meerdere nodes met *gelijke score*: we nemen de jongste, volgens het lifo-principe (last-in, first-out) zoals bij een stack.
=> de gearceerde 5 wordt genomen.

(5) A-star: voorbeeld



- Robot moet weg vinden (rood -> groen)
  - *g-score* = afstand vanaf start
  - *h-score* = afstand tot doel
  - **Laagste scores (g-score + h-score) worden eerst bekeken: hierop sorteren met PriorityQueue**
  - **Hetzelfde als best-first, enkel dat g-score wordt toegevoegd**
- http://en.wikipedia.org/wiki/A*_search_algorithm

Code breadth-first => aStar

```
public static <Zet> boolean aStar (ZoekNode<Zet> startnode){
PriorityQueue <ZoekNode<Zet>> openNodes =
                    new PriorityQueue ekNode<Zet>>(1000);
   openNodes.add(startnode);

   while(!openNodes.isEmpty()){
       ZoekNode<Zet> node = openNodes. poll()

       List<Zet> zetten = node.possibleMoves();
       for(Zet zet: zetten ){
              ZoekNode<Zet> child = node.clone(); //copy!!
              child.move(zet);
              if (child.isSolution()){
                     System.out.println("Oplossing gevonden!");
                     return true;
              }
              openNodes.add(child);
       }
   }
   return false; // geen oplossing gevonden
 }
```

Code identiek aan best-first!
Enkel score is anders

Best-first en a-star

Zoekboom p.38 in cursus en schuifpuzzel.jar

We gaan verder waar we gestopt zijn met greedy search

In vierkant: a-starscore

Als meerdere nodes met *gelijke score*: we nemen de jongste, volgens het lifo-principe (last-in, first-out) zoals bij een stack. Dus eerst de onderste 7 (bij 4), dan pas de bovenste 7 (bij 6).

p. 39

(6) Iterative Deepening

Niet in cursus

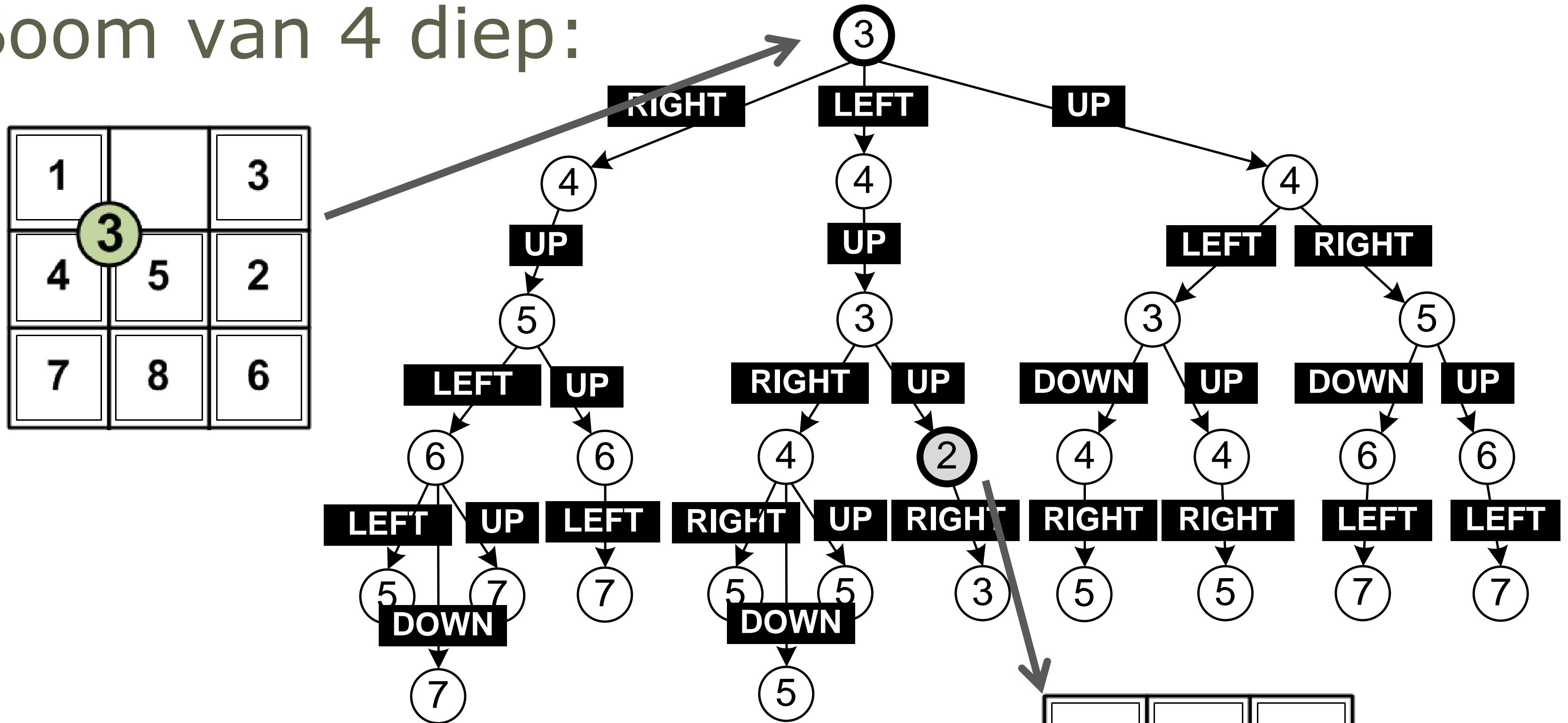
- Start met beginsituatie als beginnode
- Doe iteratief tot oplossing gevonden of maximale diepte bereikt
  - Vanaf de node de zoekboom uitwerken (breadth-first van links naar rechts) tot een zekere diepte (***horizon***)
  - Stoppen als een oplossing gevonden wordt
  - Selecteer hoogste score uit de bladeren (nodes op diepte van horizon)

    - Als verschillende nodes met dezelfde score, behouden we de eerste node die we tegenkwamen
  - Voer de zet uit die naar de hoogste score leidt
    - Deze node wordt de nieuwe beginnode voor de volgende iteratie

Je zet dus 1 stapje en werkt telkens opnieuw de zoekboom uit tot de horizon. Boom wordt dus telkens 1 level dieper uitgewerkt.

(6) Iterative Deepening

- Boom van 4 diep:



Iterative Deepening

- Boom uitwerken vanaf

Puzzel 2 in schuifpuzzel.jar
Waar greedy search stopt

1	3	2
4	5	6
7	8	

(2)

- Met diepte 4 komen we er niet…

- *In feite heeft deze puzzel helemaal geen oplossing, het verwisselen van '2' en '3' geeft een onmogelijke situatie!!*

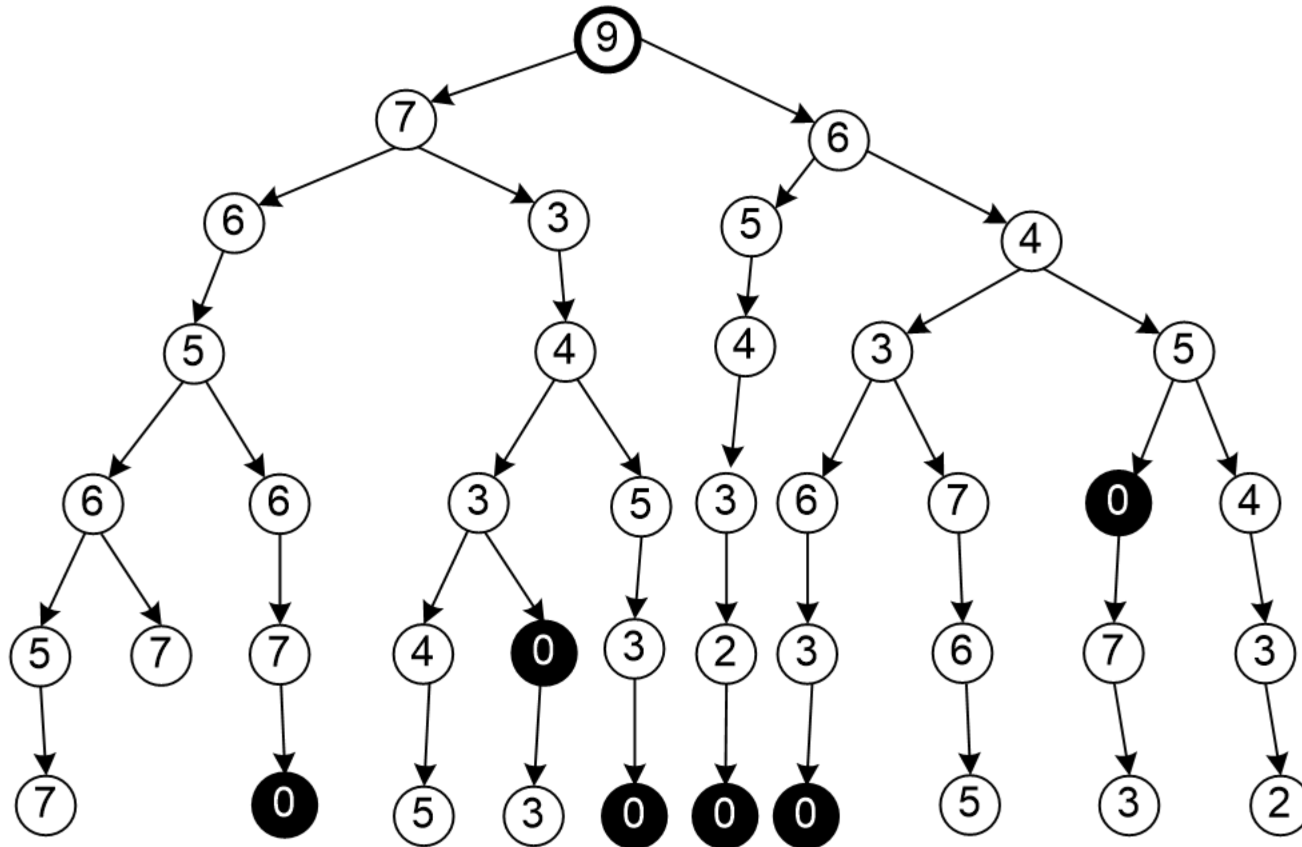
Puzzel 9 in schuifpuzzel.jar is wel een correcte puzzel

1	3	6
4	Score = 6	2
7	5	8

- Greedy search loopt vast
- Vergelijk iterative deepening met horizons 1, 2 en 3.
- Vergelijk met breadth-first en a-star

Oefening

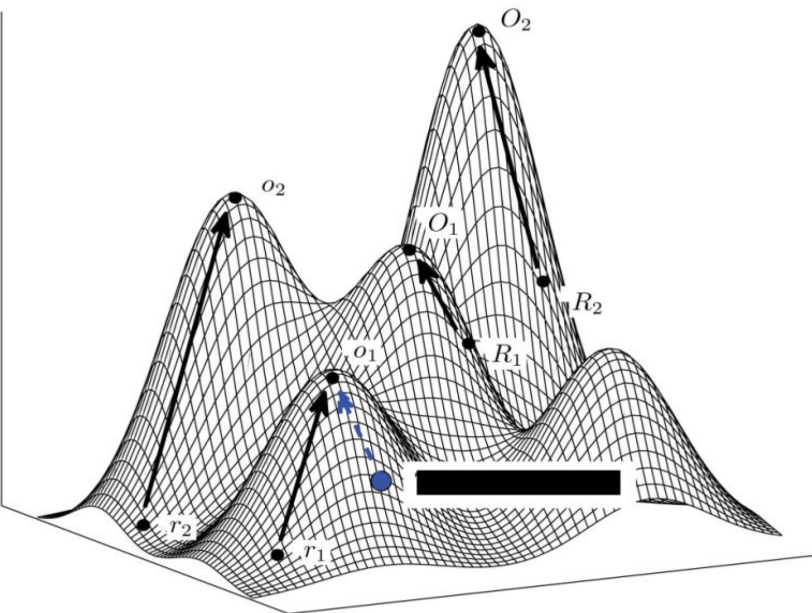
p. 40

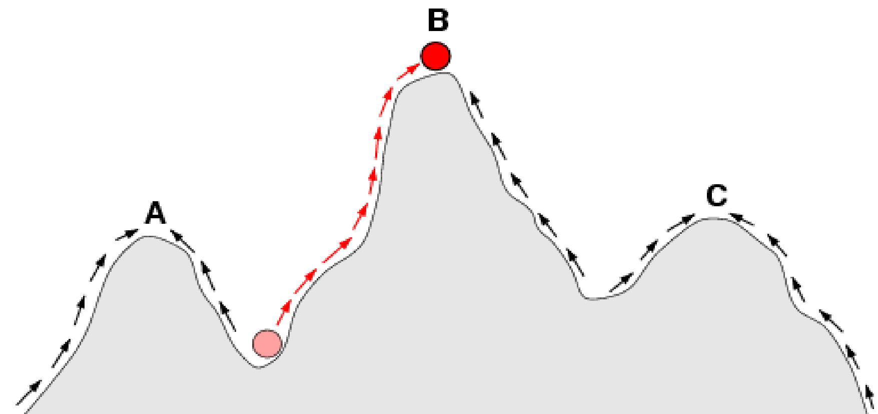


- Gegeven: zoekboom met scores (niet zeker dat een perfecte inschatting is).
- Doel: score 0
- Pas de 6 verschillende zoekalgoritmen toe (horizon=2, test ook horizon 3).
- Vergelijk de snelheid om een oplossing (zwarte node) te vinden.
- Vergelijk voor het vinden van de kortste oplossing (minst aantal stappen).
- Welke algoritmen garanderen de kortste oplossing?

Score in zoekruimte

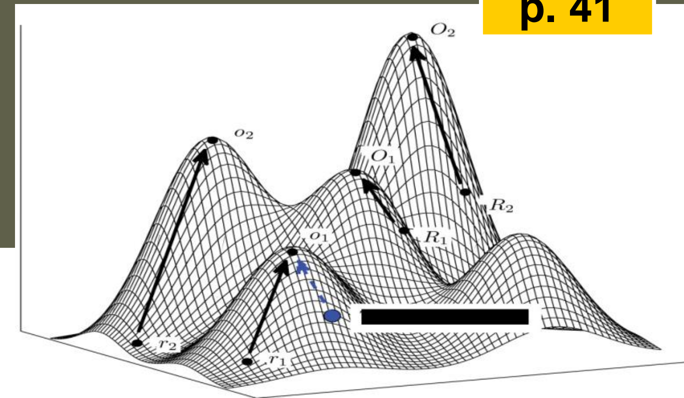
- **Probleem**: terechtkomen in **lokaal maximum**





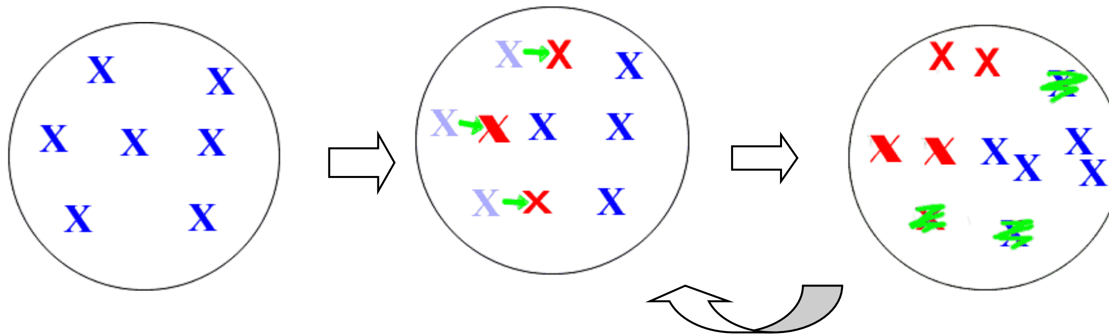
Fitness landscape

p. 41



- Biologische evolutie van soorten
  - *Fitheid:* bepaald door zijn aangepastheid aan de omgeving
  - *Natuurlijke selectie*
  - Verandering bouw en het gedrag d*oor genetische mutaties*
- Evolutie van soorten
  - Klimmen en sprongen door het fitness landscape
  - elke stabiele soort op een bergtopje zit
- Evolutie nabootsen: genetisch algoritme

Genetisch algoritme



Initële populatie ***Sommige individuën muteren*** ***Enkel de besten overleven***

p. 42

Verdere verbeteringen puzzelalgoritme

a) puzzel rij-per-rij afwerken.
- Score enkel op eerste rij definiëren, vervolgens op 2e rij, …
- Behalve de laatste twee rijen, die tegelijkertijd aanpakken
- Beperkte horizon OK

b) Leren (type 5): zie later

c) Regels opstellen
- Type 1 probleem

Hoe generiek programmeren?

niet te kennen,

optioneel – als vervangvraag

Generieke implementatie

- Via abstractie is het algoritme algemeen bruikbaar
- Code + uitleg staat op website
  - Java package '**zoeken**'
  - Niet te kennen
  - Optioneel: als vervangvraag (in de plaats van een andere vraag)

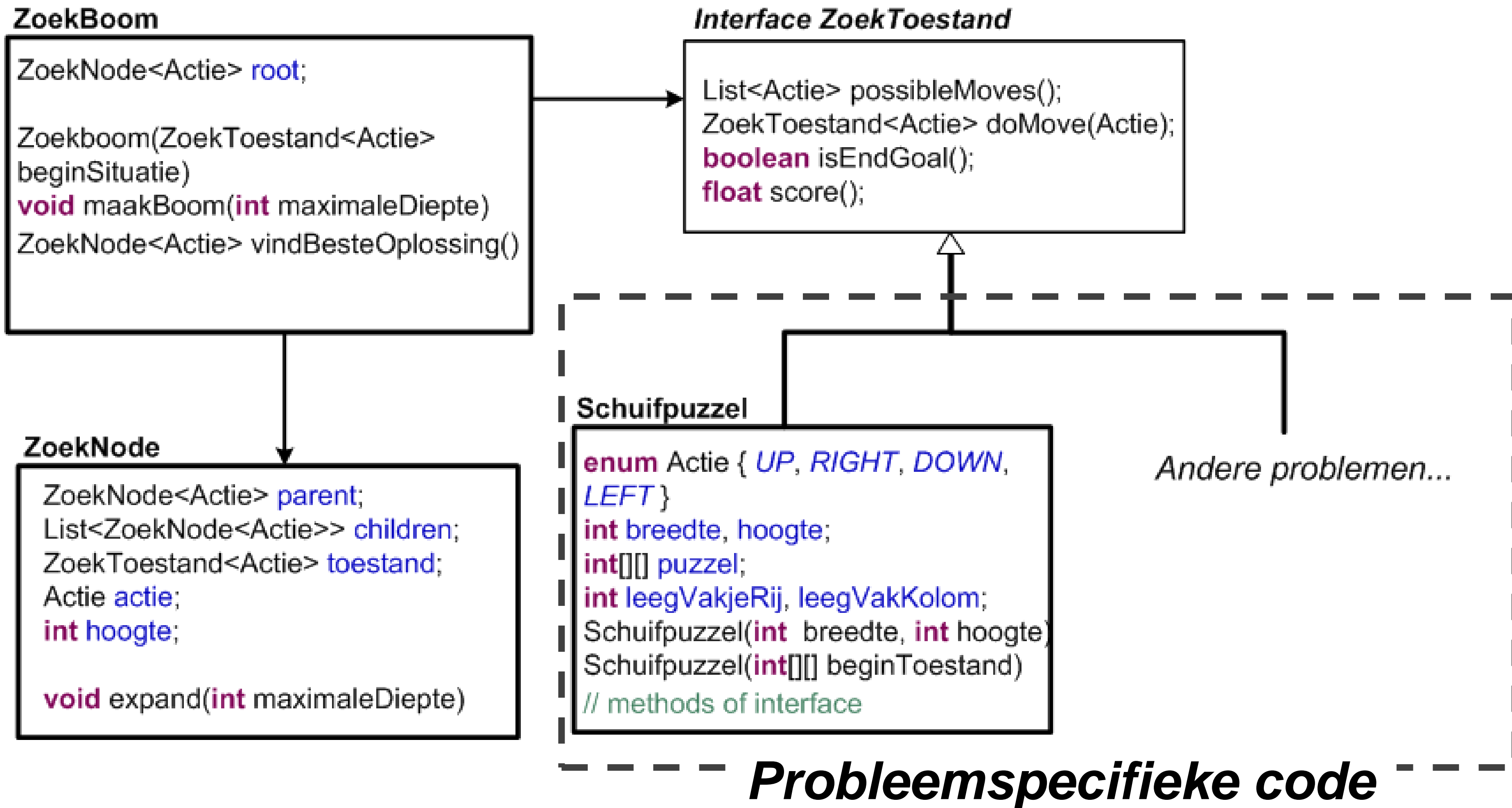
```
interface Node {
   boolean isOplossing();
   List<Zet> volgendeZetten();
   void doeZet(Zet zet);
   void ontdoeZet(Zet zet);
   Node clone();
}
interface Zet{

}
```

Uitdaging: algemene oplossing

onafhankelijk van het specifieke probleem dat je wenst op te lossen!

- Wat moet je weten om de boom aan te maken?
  Je moet de mogelijke *acties* kennen. Met de actie kom je in een nieuwe situatie, je bereikt een nieuwe toestand. Je wilt ook weten of je de gevraagde eindsituatie bereikt hebt.
  - ➡ **Allemaal operaties op de toestand**

Document op website



Link tussen algoritme en probleem gegeven door *interface*