

Inleiding

Doelstellingen

Na dit hoofdstuk

- kan je code in verschillende klassen onderbrengen
- begrijp je hoe code uit andere klassen via objecten kan aangesproken worden
- versta je het basisprincipe van het scheiden van logica en gebruikersinteractie, en waarom dit programmacode beter maakt
- ken je de voordelen van het gebruiken van een IDE voor de ontwikkeling van programmacode
- ben je vertrouwd met de basishandelingen van de NetBeans IDE
- kan je met de NetBeans debuggen en de meest courante handelingen uitvoeren zoals breekpunten zetten, door de code stappen met Step Into en Step Over, locals en watches gebruiken, etc.
- kan je met NetBeans een eenvoudige GUI bouwen en die verbinden aan je eigen code
- weet je wat events zijn en kan je een eenvoudige event listener implementeren
- kan je met packages werken en weet je hoe je eigen code erin onder kunt brengen
- kan je de Java API gebruiken en er informatie in opzoeken
- weet je hoe je een eenvoudig klassendiagramma moet interpreteren

1.1 Inleiding

In deze cursus gaan we in op wat Java een echte object-georiënteerde programmeertaal maakt. Tot nu toe hebben we **al onze code in één klasse** ondergebracht, waarin een aantal methoden voorkomen die de functionaliteit van het programma onder elkaar verdelen. Deze manier van werken is **bruikbaar voor kleine programma's**, maar **bij grotere en meer complexe code is dit niet meer werkbaar**: het wordt moeilijker om de code te **onderhouden, uit te breiden of om bestaande code te herbruiken in nieuwe projecten**. Een object-georiënteerde programmeertaal voorziet een aantal concepten en technieken om deze problemen het hoofd te bieden, om m.a.w. je programmacode beter te organiseren en te structureren. Het is de opzet van deze cursus om je een aantal van deze OO concepten bij te brengen, en meer algemeen om je een aantal goede programmeergebruiken aan te leren.

1.2 Werken met verschillende klassen

Nemen we bij wijze van voorbeeld volgend zeer eenvoudig programma, de klasse **Console**:

```
// Console.java
public class Console {
    public void start() {
        int x1, y1, x2, y2; //lokale variabelen

        System.out.print("X-coordinaat eerste punt: ");
        x1 = Input.readInt();
        System.out.print("Y-coordinaat eerste punt: ");
        y1 = Input.readInt();
        System.out.print("X-coordinaat tweede punt: ");
        x2 = Input.readInt();
        System.out.print("Y-coordinaat tweede punt: ");
        y2 = Input.readInt();

        double afstand = Math.sqrt(Math.pow(x2 - x1, 2)
                                   + Math.pow(y2 - y1, 2));
        System.out.println("De afstand is " + afstand);
    }

    → public static void main(String[] args) {
        //object van 'eigen' klasse maken en opstarten
        Console afst = new Console();
        afst.start();
    }
}
```

```

    }
}

```

In deze klasse zien we het **systeem van werken waarop we elk programma tot nu toe hebben gebouwd**; een methode **main** waarin een object wordt aangemaakt (**afst**), en via dat object wordt een methode opgeroepen (**start**) van waaruit de eigenlijke logica wordt geprogrammeerd. Zoals we weten dient de **methode main enkel als startpunt voor de toepassing**, verder heeft deze methode eigenlijk geen direct nut. Meer zelfs, eigenlijk kan die methode eender waar staan. In Java kan je perfect een programma uit meerdere klassen laten bestaan, we kunnen **main m.a.w. zonder problemen afzonderen** in een **nieuwe klasse Main**:

```

// Console.java
public class Console {
    public void start() {
        int x1, y1, x2, y2; //lokale variabelen

        System.out.print("X-coordinaat eerste punt: ");
        x1 = Input.readInt();
        System.out.print("Y-coordinaat eerste punt: ");
        y1 = Input.readInt();
        System.out.print("X-coordinaat tweede punt: ");
        x2 = Input.readInt();
        System.out.print("Y-coordinaat tweede punt: ");
        y2 = Input.readInt();

        double afstand = Math.sqrt(Math.pow(x2 - x1, 2)
                                   + Math.pow(y2 - y1, 2));
        System.out.println("De afstand is " + afstand);
    }
}

```

(gescheiden! 2 klassen ipv 1)

```

// Main.java
public class Main {
    public static void main(String[] args) {
        //object van 'andere' klasse maken en opstarten
        Console afst = new Console();
        afst.start();
    }
}

```

Het afzonderen van **main** op zich is niet bijzonder, maar het is wel een eerste kleine stap in de richting van waar we naartoe willen: **vanuit de klasse Main maken we een Console-object aan, een object behorend tot een andere klasse**. Tot nu toe

was dit anders: het object dat werd aangemaakt behoorde steeds tot de klasse waarin `main` stond, en de methode die werd opgeroepen via dat object was ook steeds te vinden in de klasse zelf. Nu zien we dat het ook mogelijk is om vanuit de ene klasse (`Main`) code op te roepen (`start`) die zich in een andere klasse (`Console`) bevindt. We hoeven m.a.w. niet al onze code in één klasse onder te brengen, we kunnen code over verschillende klassen verdelen. Het volstaat dan om de nodige objecten aan te maken en op die manier code die ergens anders staat aan te spreken.

1.3 Logica en gebruikersinteractie scheiden

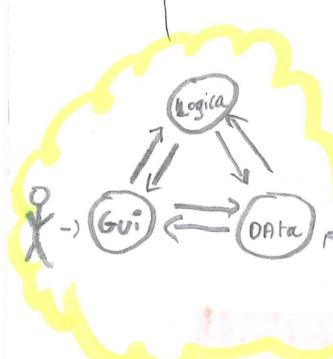
Een applicatie bestaat doorgaans uit een stuk code die de logica implementeert, ofwel de code die nodig is voor de werking van het programma, en een stuk code die de interactie met de gebruiker regelt. Het is zeer zinvol om die twee stukken (of lagen) code op te splitsen in verschillende klassen. Zo maakt de klasse (of klassen) die de gebruikersinteractie regelt gebruik van de klasse(n) waarin de logica wordt vastgelegd, en blijft de logica onafhankelijk van de keuzes die gemaakt worden voor de interactie met de gebruiker: gaat het om een console of een windows applicatie, in welke taal wordt gecommuniceerd met de gebruiker, Het kan ook zijn, zoals bv. bij web applicaties, dat deze twee programma-lagen op verschillende machines draaien. Bovendien kan er makkelijker met verschillende mensen aan één project worden gewerkt in geval het om grotere applicaties gaat. Kortom: de code wordt cleaner, overzichtelijker en makkelijker in onderhoud.

Een veel voorkomend software-model is de 3-tier architectuur of 3-lagen architectuur. In dat geval wordt er nog een extra laag toegevoegd die de communicatie met de buitenwereld (bv. het netwerk) of databron (bv. een databank) regelt. Op dit model wordt er in een latere cursus programmeren nog uitvoerig ingegaan.

Laat ons het idee van de verschillende lagen toepassen op ons eenvoudig voorbeeld. De interactie met de gebruiker bestaat uit het vragen naar coördinaten van 2 punten en het uiteindelijk weergeven van het resultaat, de afstand tussen die twee punten. De logica van ons programma bestaat uit het effectief uitrekenen van deze afstand op basis van de gegeven coördinaten. Voor de logica kunnen we volgende klasse schrijven:

```
// Punt.java
public class Punt {
    public double berekenAfstand(int x1, int y1,
                                 int x2, int y2) {
        return Math.sqrt(Math.pow(x2 - x1, 2))
```

Wegens code gesplitst; code uitwisselbaar. Person A kan aan interactie werken en person B aan functionaliteit.



```
+ Math.pow(y2 - y1, 2));
```

```
}
```

De klasse Punt bevat een methode berekenAfstand die, gegeven twee punten (x_1, y_1) en (x_2, y_2) , de afstand tussen die twee punten berekent en dat als resultaat teruggeeft. Merk op dat er in **logica-code géén IO-instructies** zullen voorkomen.

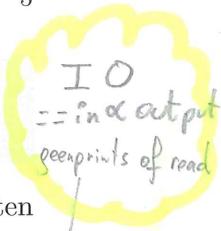
Vervolgens kunnen we de klasse Console herschrijven zodat deze gebruik maakt van onze klasse Punt:

```
// Console.java
public class Console {
    public void start() {
        Punt p = new Punt();
        int x1, y1, x2, y2;

        System.out.print("X-coordinaat eerste punt: ");
        x1 = Input.readInt();
        System.out.print("Y-coordinaat eerste punt: ");
        y1 = Input.readInt();
        System.out.print("X-coordinaat tweede punt: ");
        x2 = Input.readInt();
        System.out.print("Y-coordinaat tweede punt: ");
        y2 = Input.readInt();

        double afstand = p.berekenAfstand(x1, y1, x2, y2) ↴
        System.out.println("De afstand is " + afstand);
    }
}
```

Toegepast op ons eenvoudig voorbeeld lijkt dit nogal omslachtig, maar het legt de basis voor een idee waarop we vanaf nu al onze code gaan bouwen: de code voor de **interactie met de gebruiker** brengen we **onder in één of meerdere klassen**, en de code die de werking of de **logica van het programma** implementeert brengen we **onder in andere klassen**. Het effect zal misschien nog duidelijker worden wanneer we van deze applicatie een windows versie maken: door de scheiding in lagen moeten we enkel de gebruikersinteractie-laag in ons programma wijzigen, en blijft de rest gelijk. Maar voor we dat kunnen doen gaan we eerst een nieuwe tool introduceren die we gaan gebruiken om te programmeren: NetBeans.



1.4 De NetBeans IDE



Eenvoudige programmaatjes schrijven met een teksteditor gaat prima, maar voor het serieuzere werk zijn er tools die beter geschikt zijn. Een **IDE (Integrated Development Environment)** is een stuk software dat het leven van de programmeur een pak eenvoudiger en aangenamer kan maken. Voor het ontwikkelen van Java programma's bestaan er heel wat IDE's met NetBeans, Eclipse, IntelliJ en JBuilder als belangrijkste. Voor deze cursus gaan we gebruik maken van de NetBeans IDE (<http://www.netbeans.org/>). Dit is een ontwikkelomgeving die gratis te downloaden is voor een breed spectrum aan operating systems (Windows, Linux, Mac OS, etc.), door Sun (de ontwikkelaars van de Java programmeertaal) als IDE wordt aanbevolen en ook als download op hun eigen site beschikbaar wordt gesteld (<http://java.sun.com/javase/downloads/index.jsp>), en bovendien volledig geschreven is in Java! NetBeans is m.a.w. de logische keuze voor een eerste kennismaking met het programmeren in een moderne en volwaardige IDE.

Typische functionaliteit die je zult terugvinden in de meeste IDE's, en dus ook in NetBeans, is:

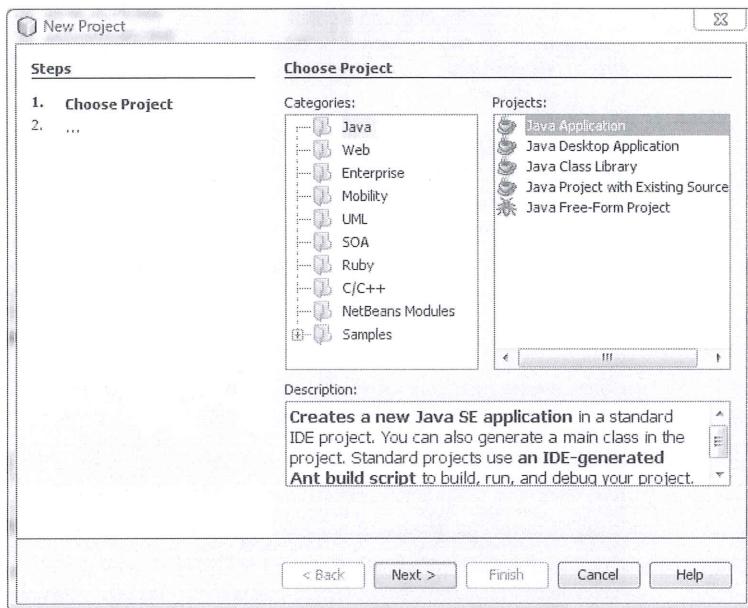
Verschill
term = kleur

- • Een geïntegreerde editor met **syntax highlighting** voor het schrijven van broncode.
- **Snel en gemakkelijk raadplegen** van de documentatie (API), o.a. door middel van **auto complete functionaliteit**.
- • **Debugger** voor het opsporen van fouten (bugs) in je programma's.
- Editor voor het **WYSIWYG** editeren van **grafische interfaces**.

What you see is what
you get!
== echte code typen

Wat volgt is een korte introductie van de NetBeans IDE en zijn gebruik. Het is belangrijk om **vooral zelf wat te experimenteren** om de mogelijkheden van NetBeans te leren kennen. Een eerste confrontatie met een IDE kan overweldigend zijn, maar eens je je draai hebt gevonden kan je niet meer zonder! Op de site van NetBeans zijn er trouwens een paar interessante tutorials te vinden die de moeite kunnen zijn om eens door te nemen (<http://www.netbeans.org/kb/>).

1.4.1 Organisatie van de workspace



Figuur 1.1: Nieuw project aanmaken in NetBeans

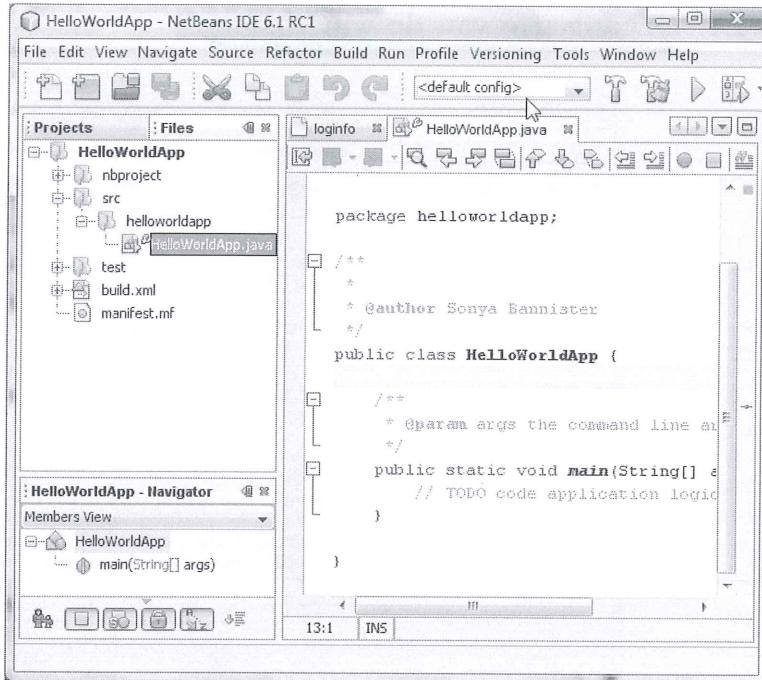
Aangezien we vanaf nu niet meer al onze code in één .java bestand gaan onderbrengen hebben we een manier nodig om verschillende klassen tot één geheel te bundelen: in NetBeans worden dit projecten genoemd. Een nieuw project maak je aan via *File > New Project* (Ctrl-Shift-N). Je kan kiezen uit verschillende project templates (zie figuur 1.1), doorgaans gaan wij werken met een gewone Java Application. Vervolgens kan je de Project Name en Project Location opgeven, en er voor kiezen om ook al een Main klasse (met de main methode) te genereren.

Eens het project is aangemaakt worden er verschillende vensters (zie figuur 1.2) geopend in de IDE (zie het menu Window mocht je er eentje verliezen onderweg):

- Het Projects venster (zie figuur 1.2 - linksboven) bevat een boomstructuur (tree view) van alle onderdelen in je project, waaronder de bronbestanden en gelinkte bibliotheken (libraries).
- In de Source Editor (zie figuur 1.2 - rechtsboven) kan je de broncode van de verschillende files bekijken.
- Via het Navigator venster (zie figuur 1.2 - linksonder) kan je snel navigeren tussen de verschillende onderdelen van je klassen. Je kan hier ook gericht de signatuur van methoden raadplegen, alsook de bijhorende javadoc.

Je kan je code compileren via de menu optie *Build > Build Main Project* (of

! N E L F U H T P R O B E R E N !



Figuur 1.2: Belangrijkste vensters in NetBeans

```

init:
deps-jar:
Created dir: C:\MyNetBeansProjects\HelloWorldApp\build\classes
Compiling 1 source file to C:\MyNetBeansProjects\HelloWorldApp\build\classes
compile:
Created dir: C:\MyNetBeansProjects\HelloWorldApp\dist
Building jar: C:\MyNetBeansProjects\HelloWorldApp\dist\HelloWorldApp.jar
To run this application from the command line without Ant, try:
java -jar "C:\MyNetBeansProjects\HelloWorldApp\dist\HelloWorldApp.jar"
jar:
BUILD SUCCESSFUL (total time: 1 second)

```

Figuur 1.3: Build Output

sneltoets F11 of het icoontje links uit figuur 1.4). De resultaten van de build kan je zien in het *Window > Output > Output* venster (zie figuur 1.3). Indien er BUILD SUCCESSFUL verschijnt waren er geen compilatiefouten, verschijnt er BUILD FAILED dan heb je minder geluk, maar kan je door te dubbel-klikken op de link snel de code te zien krijgen waar de compiler de fout in kwestie heeft gevonden. Merk echter op dat je al veel compiler-aanwijzingen krijgt tijdens het typen, dus veel compilatie-fouten kan je er al uithalen voor je effectief gaat compileren. Maar zoals steeds blijft de regel: compileer regelmatig!

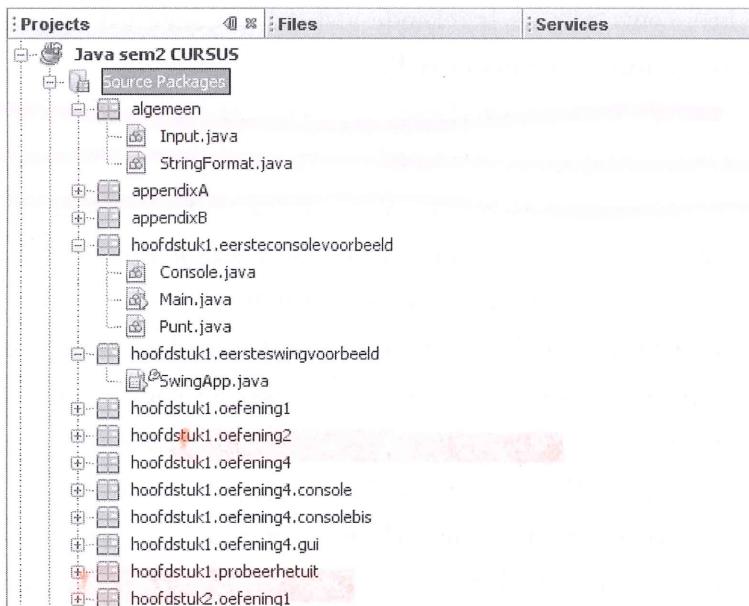
Uiteindelijk kan je je hoofdprogramma uitvoeren via de menu optie *Run > Run Main Project* (of sneltoets F6 of het icoontje rechts uit figuur 1.4). De uitvoer verschijnt eveneens in het Output venster.



Figuur 1.4: Clean and Build Main Project(F11) en Run Main Project(F6)

1.4.2 Werken met projecten

Zoals gezegd gaan we meestal werken met de Java Application template voor een nieuw project. Je kan er voor kiezen om voor je verschillende programmeeropdrachten afzonderlijke projecten te maken, maar je kan ook heel gemakkelijk je code organiseren door gebruik te maken van packages binnen eenzelfde project. Een package is een bundeling van klassen die op zich een eigen naamruimte vormen. D.w.z. dat je in principe in twee verschillende packages dezelfde klasse kunt hebben. Een package kan je heel makkelijk toevoegen aan je project door rechts te klikken op het project in het Projects venster, en te kiezen voor *New > Java Package*. Packages kan je ook nesten, een beetje zoals directories. Ze vormen een directorystructuur, met één of meerdere top-level packages (hier *algemeen*, *hoofdstuk1*, *hoofdstuk2*, ...) en daaronder de subpackages (zie figuur 1.5).



Figuur 1.5: Code organiseren d.m.v. packages

Een nieuwe Java klasse voeg je toe door rechts te klikken op je package naar keuze, en te kiezen voor *New > Java Class*. Je zult merken dat er automatisch een package instructie bovenaan de broncode wordt toegevoegd die aangeeft tot welke package deze klasse behoort, bv.

```
package hoofdstuk1.eersteconsolevoorbeeld;
```

```
public class Punt {
    ...
}
```

! Om een klasse uit een andere package te gebruiken wordt de **import** instructie gebruikt, gevolgd door de naam van het package, een . en een * (wat wil zeggen **importeer alle klassen**) of specifiek de naam van de klasse die je nodig hebt:

import java.text.DecimalFormat; importeerd van java de tekstklasse!
import algemeen.*; imp alle algemene klassen!

```
public class Console {
    ...
    // afgerond
    DecimalFormat df = new DecimalFormat("0.00");
    System.out.println("De afstand is "
        + df.format(afstand));
}
```

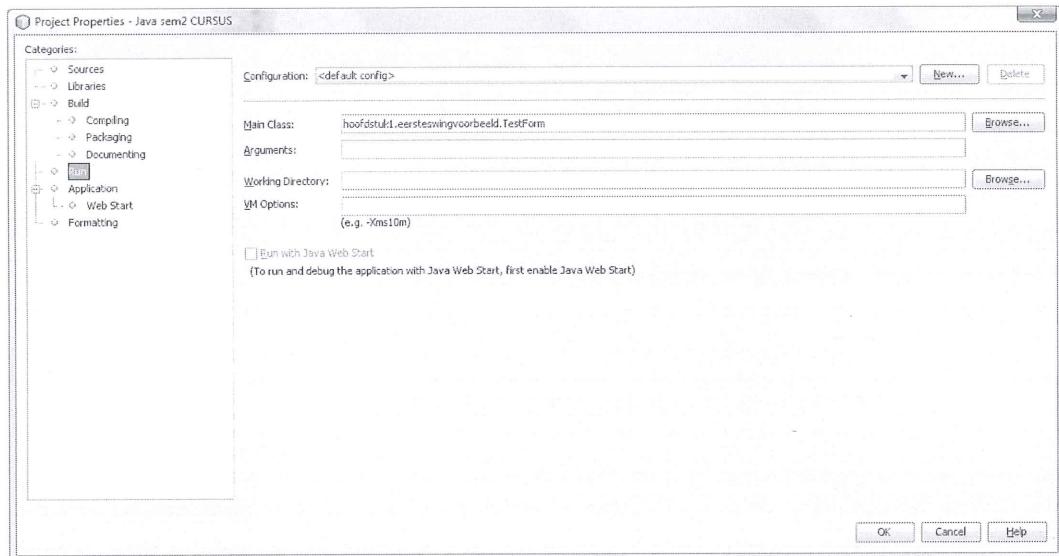
Via de hierboven aangegeven import van de Java API klasse DecimalFormat kan je heel eenvoudig de berekende afstand afronden tot op een gewenst aantal cijfers na de komma (zie appendix B.4).

De algemene Input klasse waarmee je de IO verzorgt (reeds gekend uit het eerste semester) kan je onderbrengen in een package met de generieke naam **algemeen**. Hier kan je later nog andere generieke klassen en methoden in onderbrengen.

Door gebruik te maken van packages kan je dus perfect al je laboefeningen in één project onderbrengen, zou je dat willen.

Een project in NetBeans kan gerust meerdere klassen met een **main** bevatten, door rechts te klikken op het project kan je via de Properties onder de categorie Run aanpassen **welke Main Class als startpunt** voor de build moet gekozen worden.

Met je Netbeans ontwikkelomgeving kan je meerdere projecten tegelijk maken/inladen. Door in het Projects venster een project te selecteren en via de rechtermuisknop te kiezen voor **Set As Main Project** bepaal je welk project het actieve is. Het actieve project wordt in vette druk weergegeven.



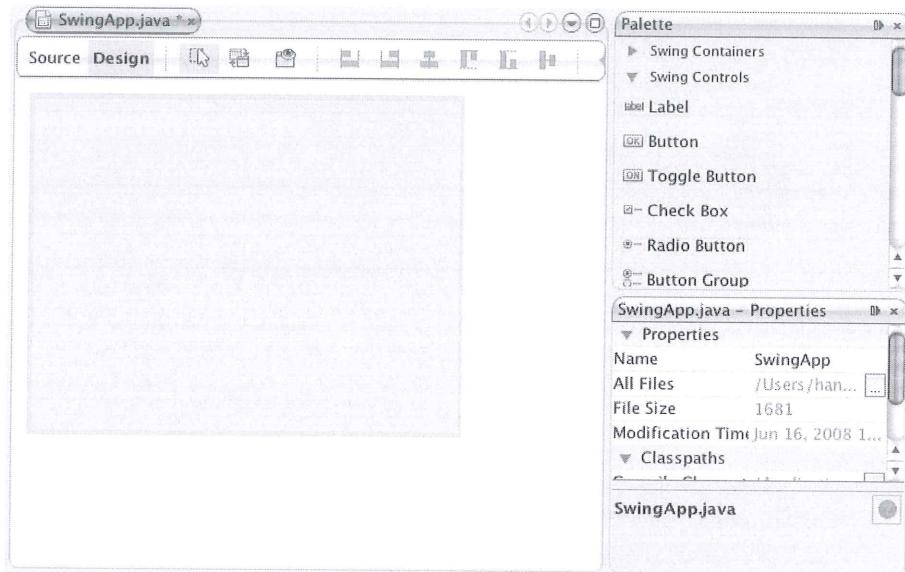
Figuur 1.6: Instellen van het startpunt van je Netbeans project

1.4.3 Een eerste GUI maken

Swing is een Java bibliotheek die gebruikt wordt voor het bouwen van Grafische User Interfaces (GUI's). Swing is gebaseerd op een andere bibliotheek, AWT of Abstract Windowing Toolkit. In tegenstelling tot AWT, waar grafische elementen van het besturingssysteem worden gebruikt, heeft Swing een eigen cross-platform library. Hierdoor biedt Swing een hogere diversiteit aan componenten aan. De meeste Swing componenten zitten in het `javax.swing` package.

NetBeans voorziet een WYSIWYG editor voor het maken van Swing GUI's. Laat ons bij wijze van voorbeeld een grafische versie maken van ons programma. Door rechts te klikken op een package en New te kiezen krijg je ook de mogelijkheid om een JFrame Form toe te voegen, dit is wat we nodig hebben en we noemen onze klasse `SwingApp`. In de workspace krijgen we hierdoor een paar nieuwe vensters te zien (zie figuur 1.7). Centraal krijg je je formulier te zien, dit komt overeen met je applicatievenster. Rechts bovenaan zie je het palet (**Palette**) met alle componenten die je via drag-and-drop aan je design formulier kunt toevoegen. Rechts onder krijg je het Properties venster met de eigenschappen van het geselecteerde component.

Via de knoppen *Source* en *Design* kan je switchen tussen ontwerpweergave (wat je standaard te zien krijgt) en de achterliggende code. Je zult merken dat zo'n **JFrame Form** klasse al uit wat code bestaat nog voor je zelf iets moet doen. De code die je te zien krijgt is de volgende (beetje uitgeknipt voor het overzicht):



Figuur 1.7: Een GUI maken

```

public class SwingApp extends javax.swing.JFrame {
    public SwingApp() {
        initComponents();
    }

    // Generated code
    private void initComponents() {
        ...

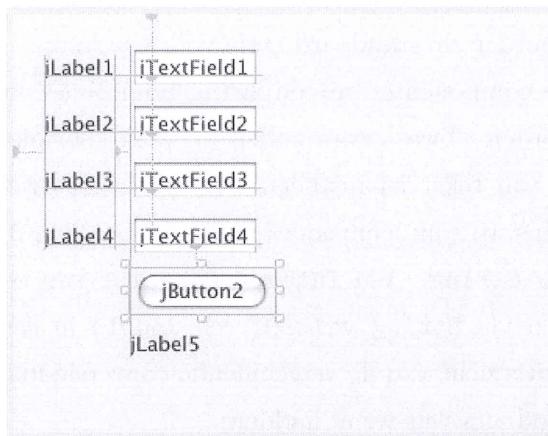
        public static void main(String args[]) {
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
                    new SwingApp().setVisible(true);
                }
            });
        }
    }
}

```

Het is niet de bedoeling om al deze code in detail te begrijpen, enkel om in grote lijnen te volgen wat er gebeurt. Zoals je kunt zien bestaat onze klasse `SwingApp` uit een `main` methode. Om deze nieuwe toepassing te laten uitvoeren selecteer je `SwingApp` als startpunt binnen je Netbeans project. Verder bestaat de klasse uit een methode `initComponents` die weggemoffeld zit in een editor-fold. Deze methode bevat code die automatisch wordt gegenereerd en nodig is voor de opbouw van ons formulier. We gaan deze code eigenlijk nooit manueel moeten wijzigen. Verder zie je nog een bijzondere methode `SwingApp`, dit is een constructor (meer hierover in

hoofdstuk 2).

Een GUI bouwen is op zich niet zo ingewikkeld. In *Design View* sleep je die componenten naar je formulier die je nodig hebt. Wij moeten 4 getallen kunnen ingeven voor de coördinaten van de 2 punten, hiervoor kan je tekstvelden (*JTextField-object*) gebruiken, met 4 labels (*JLabel-object*) erbij voor de duidelijkheid. Dan nog een knop (*JButton-object*) waarop we moeten klikken om de afstand te bepalen, en tenslotte nog een label voor het resultaat. De editor voorziet visuele hulp om het aligneren van de componenten eenvoudiger te maken (zie figuur 1.8).



Figuur 1.8: Componenten aligneren

Via het Properties venster kan je een aantal eigenschappen van de componenten op je formulier aanpassen. Via de **eigenschap title** van het *JFrame* component zelf kan je je grafische toepassing een titel geven. Met de eigenschap **text** van een component op je formulier kan je bepalen welke tekst er in of op het component wordt weergegeven. Voor elk component-object wordt ook een variablenaam gegenereerd. Die is wel conceptueel (*jTextField1*, *jTextField2*, ...), maar niet altijd functioneel eenduidig. **Het is belangrijk, zeker voor die componenten die je vanuit programmacode wil kunnen manipuleren, om een duidelijke naam te kiezen.** Wij willen ingegeven getalwaarden uit 4 tekstvelden halen, er de afstand tussen 2 punten mee berekenen en achteraf het resultaat in het onderste label weergeven. Door rechts te klikken op een component kan je via *Change Variable Name* de variablenaam van het component wijzigen. Wij noemen onze tekstvakken *jTextFieldX1*, *jTextFieldY1*, *jTextFieldX2* en *jTextFieldY2*, het label noemen we *jLabelOutput*, de knop geven we de naam *jButtonBerekenAfstand*.

Wat ons nog rest is bepalen wanneer we willen dat de afstand wordt berekend. Bij grafische applicaties wordt gewerkt met events. **Een event is een methode die uitgevoerd wordt naar aanleiding van een bepaalde actie, bv. een gebruiker die op**

vb click, of scroll, of close

een knop klikt. Je kan heel eenvoudig events toevoegen aan componenten door in *Design View* rechts te klikken op het component en onder *Events* het event in kwestie te kiezen. Er bestaan heel wat events die je kan implementeren, één van de meest gebruikte is het `actionPerformed` event van een knop - dit event zal at-runtime worden getriggerd telkens wanneer de gebruiker op die knop klikt. Door dit event toe te voegen wordt de *Source View* geopend op de plaats van de eventmethode die net werd toegevoegd. Via de *Events* tab in het *Properties* venster kan je eventueel alsnog de naam van de methode wijzigen. Merk op dat het `actionPerformed` event ook kan toegevoegd worden door gewoon te dubbelklikken op de knop. Het is met andere woorden de standaard actie voor een knop.

Op de componenten uit de *Swing* bibliotheek kunnen heel wat methoden toegepast worden. Twee ervan zullen we vaak gebruiken: `getText` en `setText`, ofwel opvragen van tekst en instellen van tekst. Let wel: deze methoden werken met `String`, dus we gaan conversies nodig hebben van `String` naar `int` en van `double` terug naar `String`. Via `Integer.parseInt` kan een `String` in een `int` omgezet worden, en via `String.valueOf` een `double` in een `String` (zie de appendix B2 voor een overzicht van de verschillende conversie-methoden). De code is verder vrij gelijklopend aan wat we al hadden:

```
private void jButtonBerekenAfstandActionPerformed(→
    →java.awt.event.ActionEvent evt) {
    int x1 = Integer.parseInt(jTextFieldX1.getText());
    int y1 = Integer.parseInt(jTextFieldY1.getText());
    int x2 = Integer.parseInt(jTextFieldX2.getText());
    int y2 = Integer.parseInt(jTextFieldY2.getText());

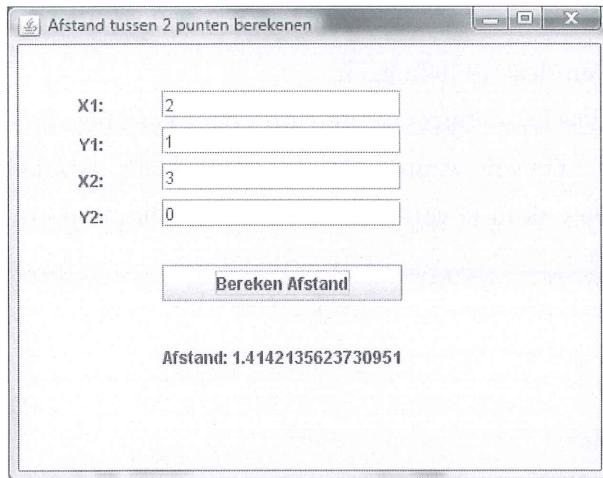
    Punt p = new Punt();
    double afstand = p.berekenAfstand(x1, y1, x2, y2);

    jLabelOutput.setText(String.valueOf(afstand));
}
```

Gewoon nog op F6 drukken en je eerste *Swing* applicatie draait! Let er op dat we aan de klasse `Punt` (de logica) niets hebben gewijzigd.

1.4.4 Debuggen

Eén van de meest interessante functies van een IDE is de mogelijkheid om je code te debuggen, wat letterlijk ontluizen betekent. Met een debugger kan je instructie per instructie door je code stappen, en kan je op elk moment de waarde van alle variabelen zien die op dat moment in scope zijn. Debugging wordt gebruikt om fouten uit programma's te halen die zich at runtime manifesteren. Zeker wanneer



Figuur 1.9: Afstand tussen twee punten - Swing style

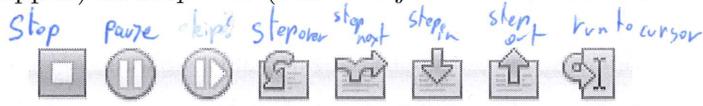
de code complexer wordt, wanneer er vaak tussen verschillende klassen weg en weer wordt gesprongen, is debugging een uiterst nuttige techniek.

Debugging in NetBeans is echt kinderspel.* Je kan de debugger starten via *Debug > Debug Main Project*. Om te bepalen waar de debugger in je code moet starten, kan je één of meerdere breekpunten zetten. Dat doe je door in de kantlijn naast de instructie in kwestie te klikken waardoor er een rood vierkantje verschijnt - er terug op klikken haalt het breekpunt weg. Als de debugger wordt gestart, wordt de code volledig uitgevoerd tot aan het eerste breekpunt.

```
public class Punt {
    public double berekenAfstand(int x1, int y1, int x2, int y2) {
        double afstand = Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));
        return afstand;
    }
}
```

Figuur 1.10: Een breekpunt plaatsen

Je kan de debugger besturen via het Debug menu of de gerelateerde shortcuts, maar handig is zeker ook de Debug toolbar (*View > Toolbars > Debug*). De interessantste knoppen zijn *Finish Debug Session*, *Continue* (lopen tot aan het volgende breekpunt), *Step Over* (waarmee je een methode kunt uitvoeren zonder in de code te stappen) en *Step Into* (waarmee je in de methode kunt stappen).



Figuur 1.11: De Debug toolbar

Kunnen
Gebruiken

Wanneer je een debug sessie start, wordt de standaard ingestelde toepassing

beschouwd. Je kan ook rechts op een bestand klikken en via de optie *Run File* vragen om deze te debuggen.

Tijdens het debuggen kan je bij Local Variables (onderaan in het Output venster) op elk moment de waarde van de verschillende variabelen volgen. Je kan tijdens het debuggen ook in je code boven variabelen hoveren om hun waarde te zien.

Watches	Local Variables	Breakpoints	Output	Tasks
Name	Type		Value	
this	Punt		#1084	
x1	int		1	
y1	int		2	
x2	int		3	
y2	int		4	
afstand	double		2.8284271247461903	

Figuur 1.12: Local Variables

1.4.5 Handig om weten

Tot slot van deze sectie over NetBeans nog een paar losse tips, kleine dingetjes die heel handig kunnen zijn en waar je soms lang kunt naar zoeken:

- Door rechts te klikken op een stuk code en te kiezen voor *Format* kan je de indentering van je code corrigeren.
- Je kan dat ook doen voor het ganse bestand via het menu: *Source > Format*
- Via *View > Show Line Numbers* kan je regelnummers aan en uit zetten.
- Je kan heel eenvoudig documentatie van een project genereren door er rechts op te klikken in het Projects venster en te kiezen voor *Generate Javadoc*.
- De Java API documentatie is ook vanuit de Netbeans omgeving toegankelijk: selecteer in het Projects venster de folder *Libraries*, klik rechts op de bibliotheek **JDK1.6 (default)** en kies de optie *Show Javadoc*.
- Door te hoveren over een signatuur in het Navigator venster krijg je de gerefereerde javadocumentatie te zien - indien beschikbaar.
- Het Navigator venster toont ook de velden van je klasse. Dit impliceert dat je hier ook alle namen opgeliist krijgt van de grafische componenten in je klasse.
- De naam van een variabele, methode, ... kan zeer gemakkelijk over de ganse toepassing aangepast worden via de *Refactor > Rename* optie.

- Eventueel ontbrekende package imports kan je achterhalen via de optie *Fix Imports*.

- !**
- Met de toetsencombinatie CTRL-SPATIE krijg je via een popup venster gerichte hulp om je code te vervolledigen.
- !**
- Je kan een volledig stuk code in commentaar zetten of uit commentaar halen met de knoppen *Comment* en *Uncomment* in de werkbalk.



Figuur 1.13: Comment en Uncomment

1.5 De Java API

De Java API (of Application Programming Interface) is de documentatie die bij het Java platform hoort. De API kan via <http://java.sun.com/j2se/1.6.0/docs/api/> geraadpleegd worden. Links bovenaan zie je alle Java packages, links onderaan de klassen binnen het geselecteerde package en in het hoofdvenster rechts de gevraagde documentatie. Een package dat een bijzondere plaats krijgt is `java.lang` aangezien het standaard aan alle Java bronbestanden wordt gelinkt. Typische klassen die je in deze package zult vinden zijn `String`, `Integer`, `Double`, `Math`, ... In de documentatie van elk van die klassen vind je o.a. een overzicht van alle methoden met hun volledige signatuur.

1.6 Probeer het uit!

Elk hoofdstuk in deze cursus gaan we besluiten met een *Probeer het Uit!*, waarin een programma stap voor stap wordt uitgewerkt. Het is sterk aanbevolen om deze programma's a.d.h.v. je cursus *zelf ook te implementeren*.

In deze sectie gaan we een grafische versie maken van het Hoger Lager spel. De computer selecteert een willekeurig getal dat de speler in maximaal 10 beurten moet proberen raden. Wanneer de speler een gok doet, geeft de computer aan of het getal hoger of lager is.

1.6.1 De logica

De logica van ons spel gaan we afzonderen in een klasse Spel. We gaan alleszins velden nodig hebben om het te raden getal en het aantal gespeelde beurten te stockeren. Via constanten kunnen we de onder- en bovengrens voor het getal vastleggen, als ook het maximaal aantal beurten.

```
// Spel.java
public class Spel {
    static final int KLEINST = 0;
    static final int GROOTST = 100;
    static final int MAX_BEURTEN = 10; } Veranderd nooit

    int teRadengetal;
    int gespeeldeBeurten; } Data storage!
```

Qua functionaliteit denken we aan een methode om het te raden getal te initialiseren. We schrijven hiervoor een methode resetSpel. Het gegenereerde getal moet tussen KLEINST en GROOTST liggen.

```
public void resetSpel() {
    teRadengetal = (int)((GROOTST - KLEINST + 1)
        * Math.random() + KLEINST); //getal tussen min-max
}
```

Een andere methode die we nodig zullen hebben is er eentje om de gok van de speler te verwerken: de gok kan te laag zijn, te hoog, correct of ongeldig. De code zou er als volgt uit kunnen zien:

```
public String doeGok(int gok) {
    if (gok < KLEINST || gok > GROOTST) { } buiten min-max
        *return "Ongeldig"; }

    gespeeldeBeurten++;

    if (gok < teRadengatal) { } foutief
        *return "Hoger"; }

    if (gok > teRadengatal) { } geldig
        *return "Lager"; }

    *return "Correct"; } jist
```

! beurten incrementeerd enkel bij geldig!
! return == "exit"-punten *

Op het eerste zicht lijkt dit ok, maar het kan beter. Door methoden in de logica-laag **tekst te laten teruggeven** bepaal je eigenlijk al te veel, zoals de taal en wat die tekst zal zijn. Een beter piste is om met getallen te werken. We kennen aan alle mogelijke resultaten van de methode getallen toe, en voor elk van die getallen voorzien we constanten. Zo krijgt de oproepende code dezelfde feedback en kan ze nog volledig bepalen wat de output moet zijn. **Merk op dat de getallen op zich er eigenlijk niet toe doen.** => gewoon afspraak dat waarde betekent

```
static final int ONGELDIG = -1;
static final int CORRECT = 0;
static final int HOGER = 1;
static final int LAGER = 2;

public int doeGok(int gok) {
    if (gok < KLEINSTEN || gok > GROOTSTE) {
        return ONGELDIG;
    }

    gespeeldeBeurten++;

    if (gok < teRadengetal) {
        return HOGER;
    }

    if (gok > teRadengatal) {
        return LAGER;
    }

    return CORRECT;
}
```

We vervolledigen onze klasse **Spel** met nog drie methoden om het aantal res-
terende beurten, het aantal gespeelde beurten en het te raden getal op te vragen.
Eigenlijk kunnen we hiervoor ook gewoon de waarde van de velden opvragen, maar
het is **beter om methoden te voorzien** (zie hiervoor ook het volgende hoofdstuk). De
volledige klasse ziet er dan als volgt uit:

```
// Spel.java
public class Spel {
    static final int KLEINSTEN = 0;
    static final int GROOTSTE = 100;
    static final int MAX_BEURTEN = 10;

    static final int ONGELDIG = -1;
    static final int CORRECT = 0;
    static final int HOGER = 1;
```

Veiligheid.
Zo kan het veld niet
beïnvloed worden.
kan er extra eindberekening
gebeuren etc...

```

static final int LAGER = 2;

int teRadengetal;
int gespeeldeBeurten;

public void resetSpel() {
    ... zie voorig
}

public int doeGok() {
    ... zie voorig
}

public int geefResterendeBeurten() {
    return MAX_BEURTEN - gespeeldeBeurten;
}

public int geefGespeeldeBeurten() {
    return gespeeldeBeurten;
}

public int geefTeRadengatal() {
    return teRadengatal;
}
}

```

Logica is af - nu kan er een console/GUI uitbreiding op gebouwd worden. Logica is af!

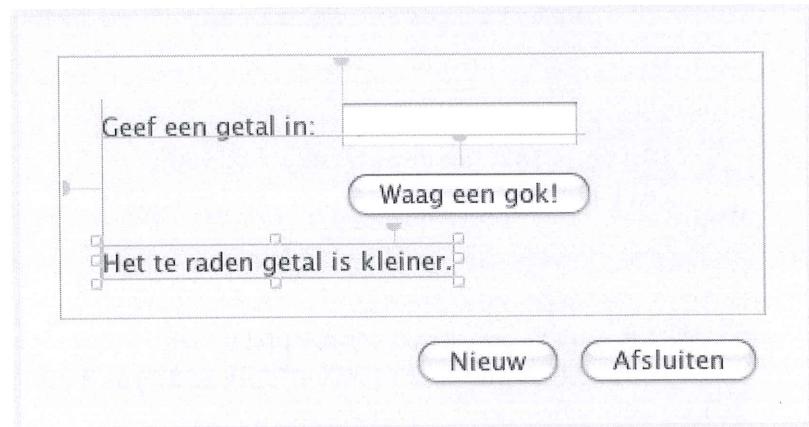
1.6.2 De GUI

Nu zijn we klaar om bovenop de klasse Spel een GUI te bouwen. We voegen hiervoor een JFrame aan het project toe. Denk er aan dat je via de eigenschappen van het project kunt bepalen wat de Main Class is. Het ontwerp van de GUI is vrij eenvoudig: via een Text Field geeft de speler z'n gokken in, klikt op de knop en in een Label verschijnt de feedback.

Onze JFrame hebben we HogerLager genoemd. In deze klasse gaan we natuurlijk een Spel object nodig hebben om alle methoden uit de klasse Spel te kunnen gebruiken. Om er voor te zorgen dat we dit object vanuit alle knop-events kunnen gebruiken maken we er een veld van. Een goede plaats om dit veld te initialiseren is in de methode HogerLager. Je kan aan de () zien dat het hier om een methode gaat, weliswaar een speciale methode die we een constructor noemen, we komen hier in het volgende hoofdstuk op terug.

maakt een obj van een klasse.
klasse = blueprint
obj = gemaakt "ding" op basis blueprint

Vb
Gvi



Figuur 1.14: De Hoger Lager GUI

```
// HogerLager.java
public class HogerLager extends javax.swing.JFrame {
    Spel spel;

    public HogerLager() {
        initComponents();

        // maak het Spel object aan
        spel = new Spel();

        // initialiseer het te raden getal
        spel.resetSpel();
    }

    ...
}
```

De **invulling van de verschillende knop-events is vrij triviaal.** Merk op dat we het tekstveld jTextFieldInput en het label jLabelOutput hebben genoemd. We geven hier de code:

```
private void jButtonGokActionPerformed(...) {
    int gok = Integer.parseInt(jTextFieldInput.getText());
    int result = spel.doeGok(gok);
    int teRadengetal = spel.geefTeRadengetal();
    int restBeurten = spel.geefRestendeBeurten();
    int gespBeurten = spel.geefGespeeldeBeurten();

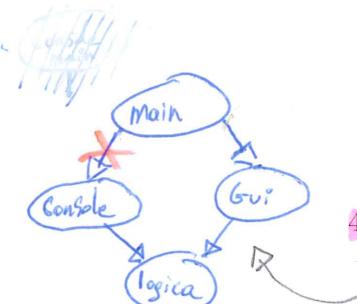
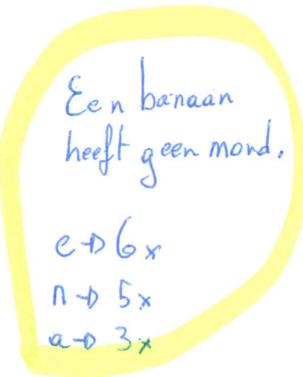
    // maximaal aantal beurten bereikt
    if (restBeurten == 0) {
        jLabelOutput.setText("Helaas, je beurten zijn op.");
    }
}
```

```
    → Het te raden getal was " + teRadengetal + ".  
    →");  
  
    // disable de gok-knop  
    jButtonGok.setEnabled(false);  
    return;  
}  
  
// verwerk resultaat  
if (result == Spel.ONGELDIG) {  
    jLabelOutput.setText("Ongeldige ingave.");  
}  
else if (result == Spel.HOGER) {  
    jLabelOutput.setText("Hoger!");  
}  
else if (result == Spel.LAGER) {  
    jLabelOutput.setText("Lager!");  
}  
else {  
    jLabelOutput.setText("Proficiat! Je hebt het  
    →getal geraden in " + gespBeurten + " beurten.  
    →");  
    jButtonGok.setEnabled(false);  
}  
}  
  
private void jButtonNieuwActionPerformed(...) {  
    spel.resetSpel();  
    output.setText("");  
    jButtonGok.setEnabled(true);  
}  
  
private void jButtonExitActionPerformed(...) {  
    System.exit(0);  
}
```

afsluitend prompt "Zeker?"

1.7 Oefeningen

Begin een nieuw Java Application project in NetBeans. Verzorg zoals steeds je programmeerstijl, en voorzie elke klasse van javadocumentatie.



- Schrijf een console toepassing die hoeveel keer een gegeven letter in een gegeven tekst voorkomt. Maak, zoals vorig semester, gebruik van de Input klasse om de interactie met de gebruiker te verzorgen. Je toepassing zelf wordt opgedeeld in 3 klassen: de klasse Main met een methode main, de klasse Console met een methode start en de klasse MijnString met een methode telLetter. De methode telLetter wordt vanuit de methode start opgeroepen. De methode start wordt vanuit de methode main opgeroepen.
Scanner


```

graph TD
    main[main] --> console[Console]
    console --> logica[Logica]
    logica --> main

```
- Genereer javadocumentatie (de html pagina's) voor alle klassen van je consoletoepassing.
- Plaats een breekpunt op de eerste instructie die wordt uitgevoerd in de code uit oefening 1. Start de debugger en doorloop je code stap voor stap met Step Into. Bekijk wat er gebeurt in het Local Variables venster. Haal het vorige breekpunt weg, en plaats er nu een op de eerste instructie van de methode die je uit de Console klasse oproept. Start opnieuw de debugger en stap nu door je code met Step Over.
- Maak van de vorige oefening een grafische versie. De gebruiker moet via tekstvelden een tekst en een letter ingeven, op een knop klikken en het aantal voorkomens van die letter verschijnt in een label. Maak gebruik van dezelfde klasse MijnString.
- Baseer je op de versie van Hoger Lager uit dit hoofdstuk om een omgekeerde versie te maken: als speler moet je een getal tussen 0 en 500 in je hoofd houden, de computer krijgt maximaal 10 beurten om dit getal te raden en als speler moet je de computer op weg helpen. Om de zoekstrategie van de computer te implementeren moet je jezelf eens afvragen hoe je zelf te werk gaat om hoger lager te spelen. Bedenk ook een goeie manier om de speler hoger, lager en correct te laten ingeven. Breng zoveel mogelijk code onder in je logische klasse. Definieer je velden in deze klasse privaat.