Informatik 1 mit BlueJ

Ein Kurs für die Stufe
EF (10)

von Ulrich Helmich

Teil 1

Folgen 1 bis 9

Aktueller Stand: 28. April 2015

Kapitelübersicht

Folge 2 - Wir programmieren 32

Folge 3 - Sie werden gewogen! 40

Folge 4 - Schleifen 82

Folge 5 - Java-Applets 103

Folge 7 - Arrays 173

Folge 8 -Sortierverfahren 195

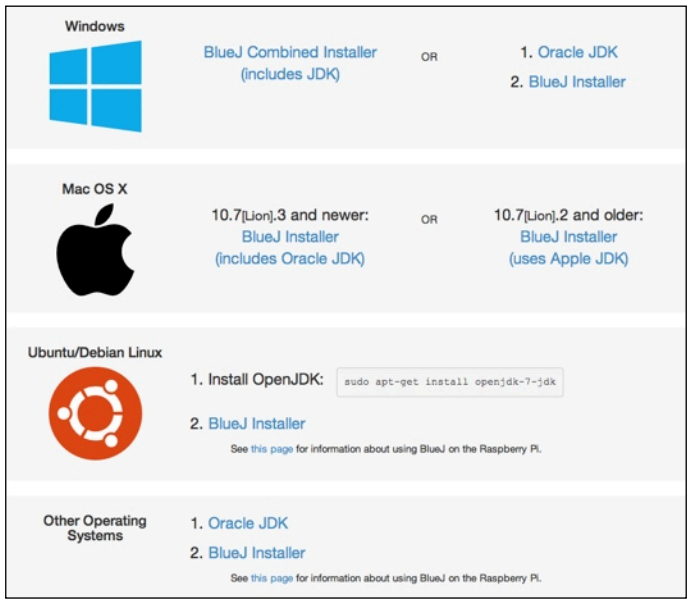
Folge 9 - Zweidimensionale Arrays 214

Folge 1 -BlueJ

Unterrichtsvorhaben

1.1 Installation und Einrichtung von BlueJ (fakultativ)

Wenn Sie mit diesem Skript arbeiten und Java lernen wollen, müssen Sie die Java-Entwicklungsumgebung BlueJ auf Ihrem Rechner installieren, falls das noch nicht der Fall ist. Dazu gehen Sie auf die Webseite www.BlueJ.org und laden sich die für Ihr Betriebssystem passende Version herunter.



***1.1 - 1** Ausschnitt aus der Startseite von www.BlueJ.org*

Vor einigen Jahren hätten Sie sich auch noch um die Installation des jeweils aktuellen JDK (Java Developement Kit) kümmern müssen. Heute ist das nicht mehr nötig. Die BlueJ-Webseite bietet Ihnen einen Komplett-Download für Ihr Betriebssystem an. In diesem "BlueJ Combined Installer" ist das JDK bereits enthalten.

Wenn Sie das Skript in Ihrer Schule durcharbeiten wollen bzw. müssen, hat der Systemadministrator normalerweise das JDK und eine aktuelle BlueJ-Version installiert, so dass für Sie dieser erste Schritt entfällt.

Das schöne an dem Programm BlueJ ist, dass man es ohne Probleme auf einen USB-Stick oder eine andere Festplatte kopieren kann. Wenn Sie BlueJ auf einen Speicher mit vollem Schreibzugriff kopieren, können Sie zahlreiche Einstellungen ändern, zum Beispiel die Sprache von Englisch auf Deutsch umstellen, eigene Templates (Vorlagen) erstellen und vieles mehr.

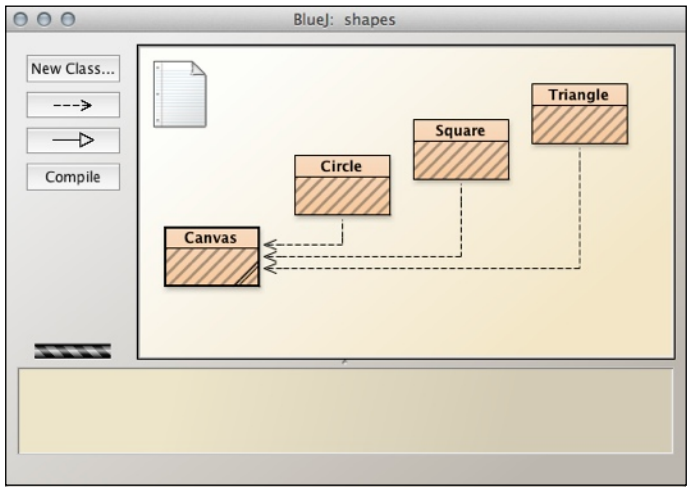
Unterrichtsvorhaben

1.2 Kennenlernen von BlueJ

Schritt 1 - Projekt "shapes" öffnen

Ihr frisch installierter BlueJ-Ordner enthält einen Unterordner "examples". Dieser Unterordner enthält Beispiel-Projekte, und mit einem dieser Projekte wollen wir uns jetzt näher beschäftigen.

➪Wählen Sie den Menü-Befehl ***Projekt öffnen*** aus dem BlueJ-Menü ***Projekt***. Er erscheint eine Liste von acht Projekten. Wählen Sie hier das Projekt "shapes" und klicken Sie dann auf ***Öffnen***.



1.2-1 *Das Beispiel-Projekt "shapes"*

So wie in der Abbildung müsste Ihr Projekt am Ende von Schritt 1 jetzt aussehen.

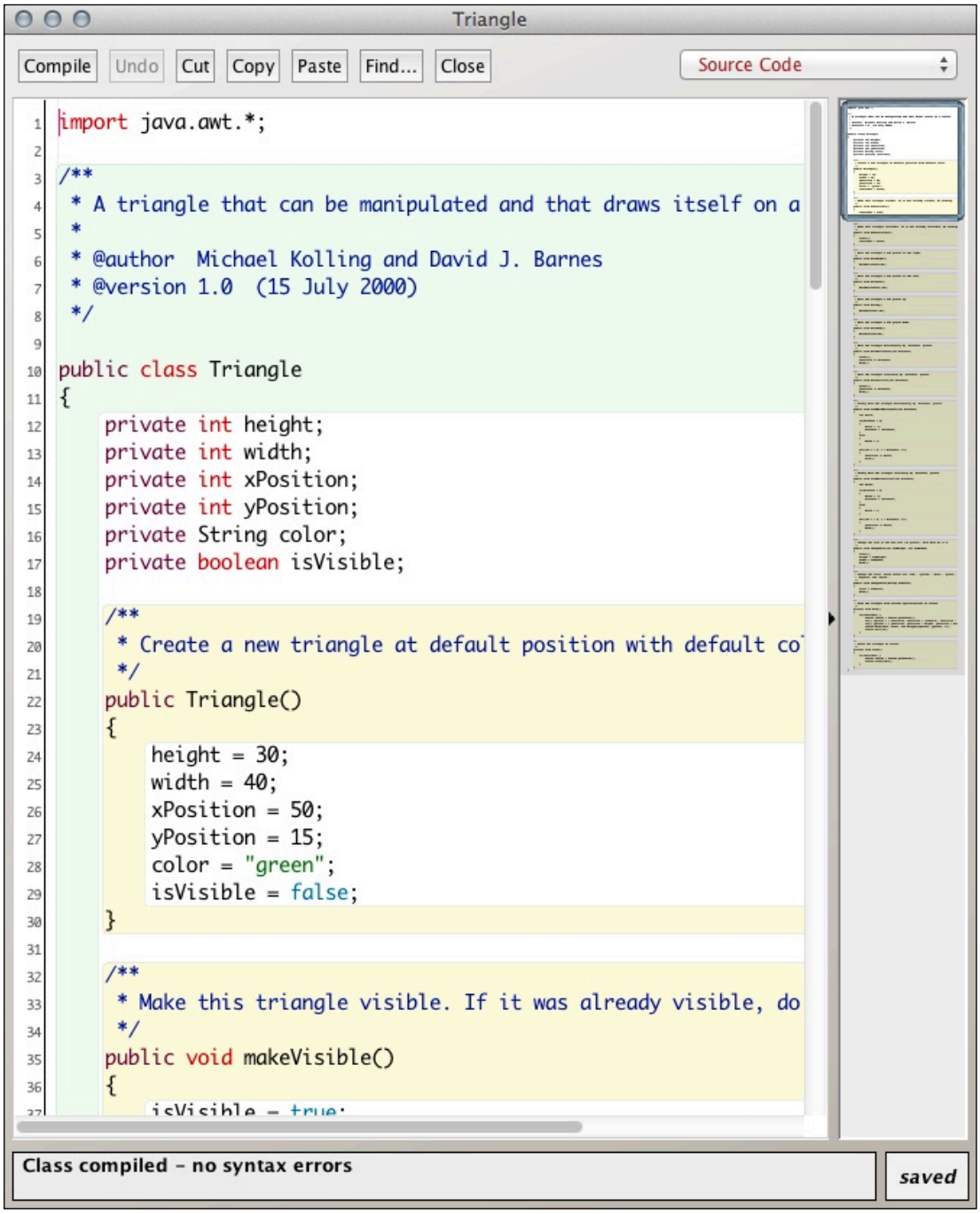
Das BlueJ-Fenster besteht aus drei Bereichen. Links sehen Sie eine Art "Bedienbereich", hier sind wichtige **Buttons** wie ***New Class*** und ***Compile*** angeordnet. Den Hauptarbeitsbereich kann man fast nicht übersehen, er enthält im Augenblick die Symbole der vier **Klassen** **Canvas**, **Circle**, **Square** und **Triangle** sowie das Symbol einer Datei, in der Sie Notizen zum Projekt und zu den Klassen unterbringen können.

Der untere Bereich ist zur Zeit noch leer, hier erscheinen später die **Objekte**, mit denen wir dann arbeiten werden.

Schritt 2 - Quelltext!

Was machen wir jetzt mit dem Projekt? Zunächst spielen wir mal ein bisschen herum.

➪Führen Sie einen Doppelklick auf die Klasse **Triangle** aus.

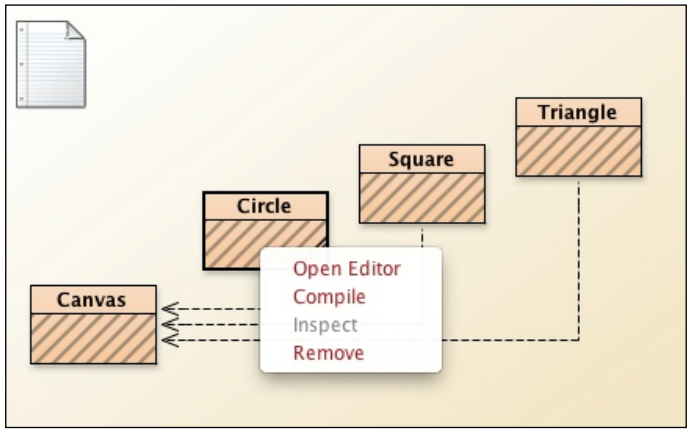


***1.2-2** Hilfe, ein Java-Quelltext!!!*

Du meine Güte, was ist das denn? Offensichtlich ein **Java-Quelltext**. Aber damit wollen wir uns doch in den ersten Unterrichtsvorhaben noch gar nicht beschäftigen. Also schnell das Fenster wieder schließen! Ja, auch Sie da hinten!

Schritt 3 - Klassen

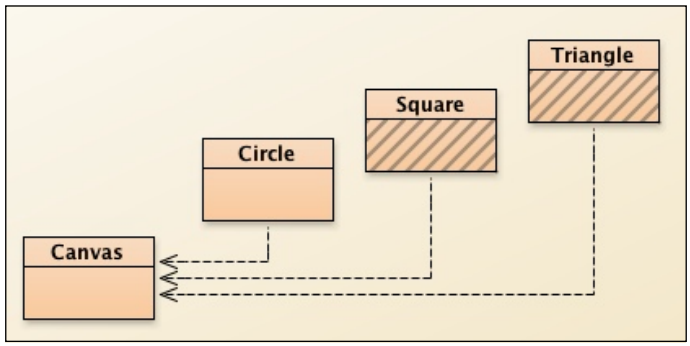
➪ Klicken Sie mit der *rechten* Maustaste auf den Kasten, der für die Klasse **Circle** steht.



1.2-3 *Nach einem Rechts-Klick auf "Circle"*

Ein Quelltext-Fenster erscheint zum Glück nicht, aber dafür ein so genanntes **Kontextmenü** mit vier **Befehlen**, von denen einer *deaktiviert* ist. Der für uns wichtigste Befehl ist ***Compile*** (falls Sie mit deutschen Einstellungen arbeiten: ***Übersetzen***).

➪ Führen Sie den Compile-Befehl aus und beobachten Sie, was sich im Arbeitsbereich ändert:

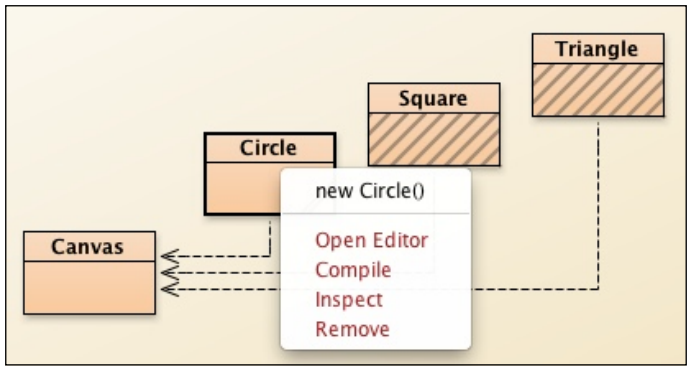


1.2-4 *Nach Ausführen des Befehls "Compile"*

Die beiden unteren Kästchen - "Canvas" und "Circle" sind nicht mehr schraffiert, während die beiden oberen Kästchen sich nicht verändert haben. Der ***Compile***-Befehl hat also etwas bewirkt. Was dabei genau passiert ist, werden Sie im nächsten Schritt erfahren.

Schritt 4 - Kompilieren einer Klasse

➪Klicken Sie noch einmal mit der *rechten* Maustaste auf den Kasten der Klasse **Circle**:



1.2-5 *Rechtsklick auf "Circle", nachdem der Befehl "Compile" ausgeführt wurde*

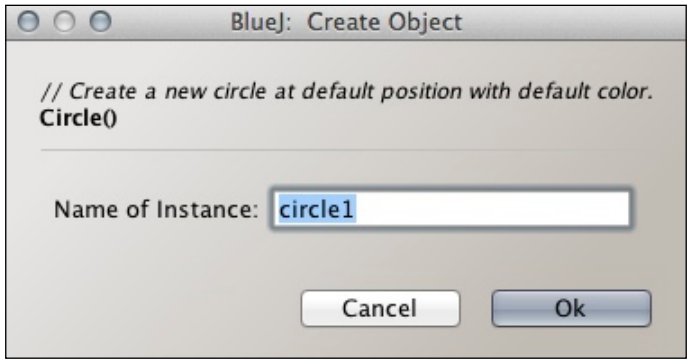
Nach dem Kompilieren stehen *neue Befehle* zur Verfügung, zum Beispiel ***new Circle()***.

Was passiert, wenn man mit der Maus diesen Befehl auswählt?

Schritt 5 - Objekt erzeugen

➪Führen Sie den Befehl ***new Circle()*** aus!

Es erscheint sofort eine **Dialogbox**:

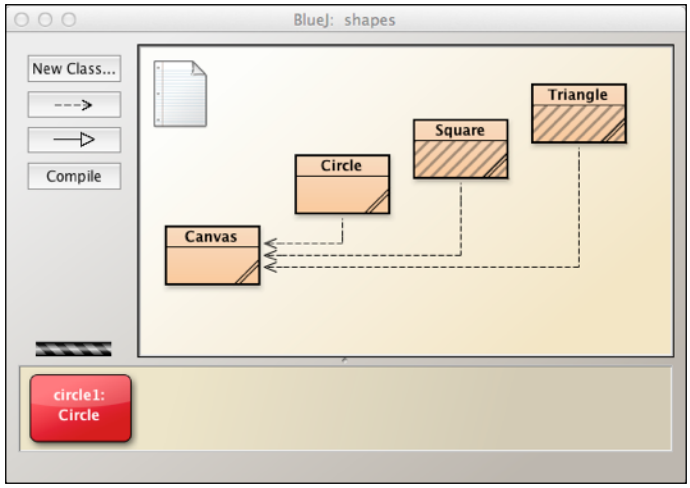


1.2-6 *Die Wirkung des Befehls "new Circle"*

Den vorgegebenen Namen "circle1" ändern wir erst einmal nicht, sondern übernehmen ihn durch einen Klick auf den Button.

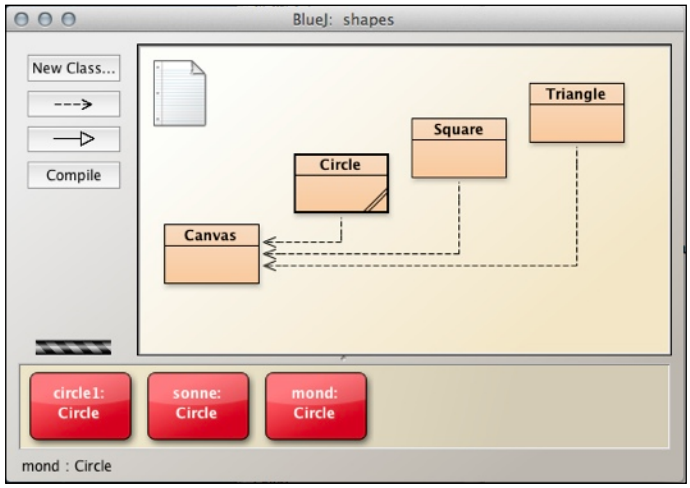
➪Klicken Sie auf den ***Ok-Button***.

Um zu sehen, was der Befehl ***new Circle()*** bewirkt hat, schauen wir uns das BlueJ-Fenster an:



1.2-7 *Das BlueJ-Fenster nach dem Ausführen des Befehls "new Circle()".*

Wir haben ein **Objekt circle1** erzeugt. Auf gleiche Weise kann man weitere Objekte erzeugen:



1.2-8 *Drei Circle-Objekte wurden erzeugt*

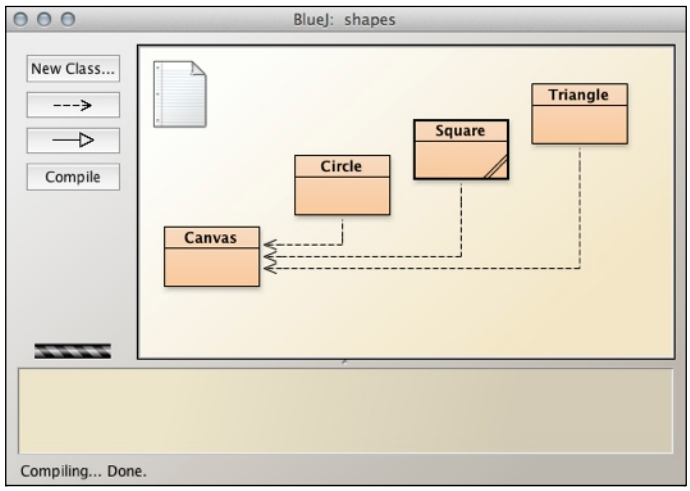
In der Abbildung 1.2-8 sieht man die drei **Circle**-Objekte **circle1**, **sonne** und **mond**. Objekte werde ich übrigens in diesem Skript künftig in **rot** schreiben. Auch für **Klassen** und **Methoden** habe ich eine eigene Farbe gewählt, um die Übersicht zu erhöhen. Ich gehe davon aus, das die meisten von Ihnen sich dieses Skript in Form einer PDF-Datei auf dem Monitor durchlesen werden.

Schritt 6 - Weitere Objekte erzeugen

➪Erzeugen Sie nun ein Objekt der Klasse **Square**.

➪Klicken Sie mit der rechten Maustaste auf den Kasten **Square** und führen Sie den Befehl ***Compile*** aus. Dadurch verschwindet die Schraffur des Kastens.

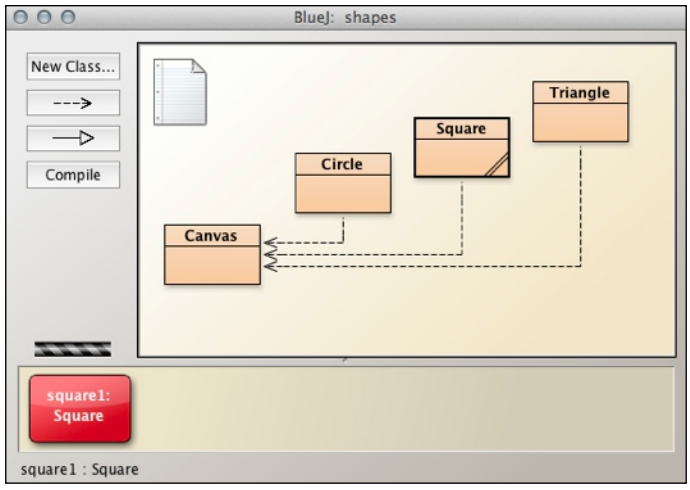
Aber gleichzeitig passiert etwas anderes:



1.2-9 *Wo sind die Objekte?*

Die drei erzeugten **Circle**-Objekte sind verschwunden! **Objekte werden nicht gespeichert.**

➪Rufen Sie nun den ***new Square()***-Befehl auf, um ein neues **Square**-Objekt zu erzeugen:



1.2-10 *Ein Objekt ist wieder da!*

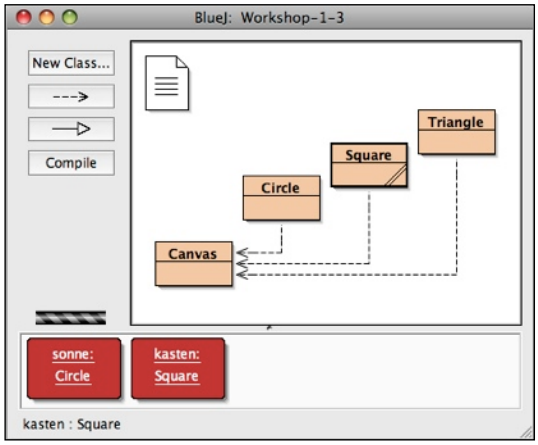
Damit haben wir den ersten Workshop bzw. das erste Unterrichtsvorhaben beendet. Programmiert haben wir noch nicht, auch über Klassen und Objekte haben wir noch nicht gesprochen, aber wir wollen ja auch nichts überstürzen, wir haben noch viel Zeit, das alles zu verstehen.

Unterrichtsvorhaben

1.3 Mit BlueJ ein Bild malen (I1 / M)

Schritt 1 - zwei Objekte erzeugen

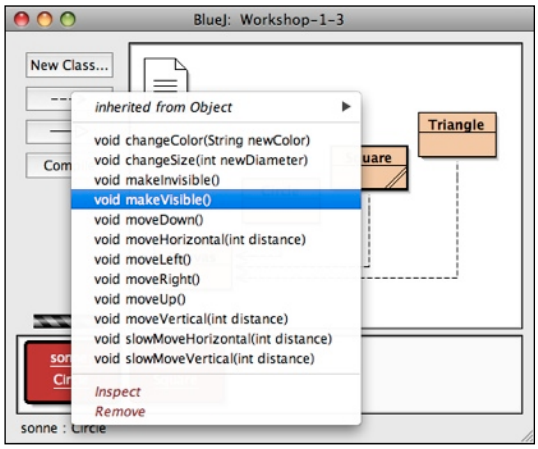
➪Erzeugen Sie ein Objekt **sonne** der Klasse **Circle** sowie ein Objekt **kasten** der Klasse **Square**.



***1.3-1** Erzeugen von Objekten einer Klasse*

Schritt 2 - Methoden ansehen

➪Klicken Sie mit der *rechten* Maustaste auf den roten Kasten des Objektes **sonne**.

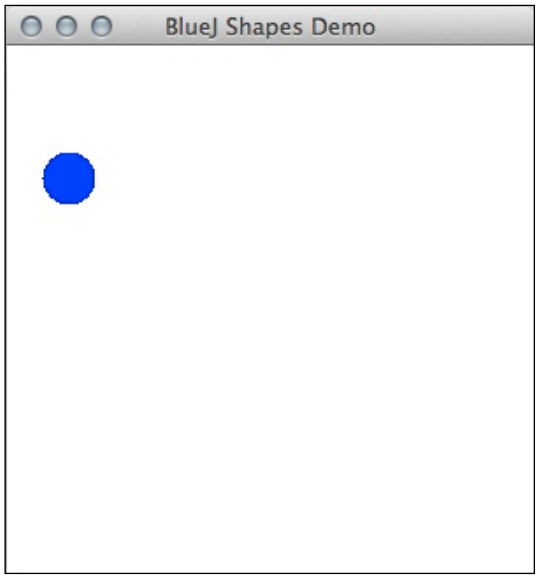


***1.3-2** Die Methoden eines Circle-Objektes*

Es erscheint ein sehr langes Kontextmenü mit vielen Einträgen. Die beiden unteren Einträge sind **Befehle**, wie Sie sie bereits kennen gelernt haben: ***Inspect*** und ***Remove***. Rufen wir den Befehl ***Inspect*** auf, so können wir das Objekt **sonne** in Ruhe untersuchen. Und wenn wir den Befehl ***Remove*** aufrufen, wird das Objekt **sonne** wieder entfernt. Die vielen anderen Einträge sind sogenannte **Methoden**; mit diesen Methoden werden wir uns in den nächsten Schritten intensiv beschäftigen.

Schritt 3 - Objekt sichtbar machen

➪Klicken Sie mit der *rechten* Maustaste auf das Objekt **sonne** und rufen Sie die Methode **makeVisible()** auf.



1.3-3 *Ein sichtbar gemachtes Objekt der Klasse Circle*

Das Ergebnis dieses Methoden-Aufrufs sehen Sie in Abbildung 1.3-3. Es erscheint ein neues Fenster mit weißem Hintergrund. In diesem Fenster wird ein blauer Kreis angezeigt.

Sie haben durch den Aufruf der Methode **makeVisible()** den **Zustand** des Objektes **sonne** verändert: Vorher war das Objekt nicht sichtbar, jetzt ist es in dem Fenster zu sehen.

Methoden

Methoden sind Befehle, mit denen man Objekte verändern kann.

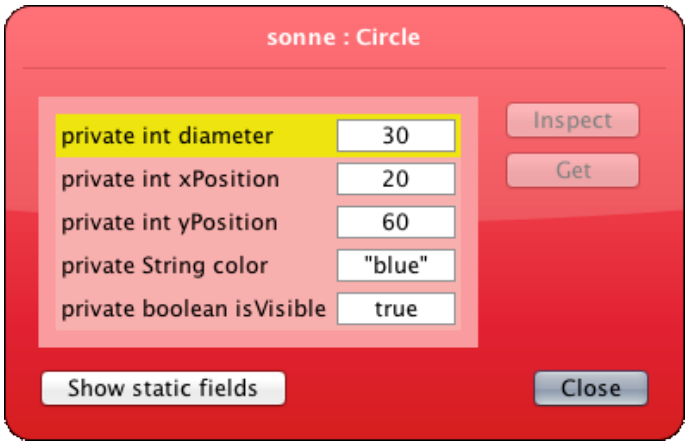
Diese provisorische Definition werden wir später noch erheblich erweitern, für's erste sollte sie aber mal genügen.

Schritt 4 - die Attribute eines Objekts

➪Erstellen Sie ein **Circle**-Objekt und klicken dann mit der *rechten* Maustaste auf den roten Kasten, der das Objekt repräsentiert. Wählen Sie in dem Kontextmenü den Befehl ***Inspect***.

Inspect ist übrigens *keine Methode* der Klasse **Circle**, sondern ein Befehl der Java-Entwicklungsumgebung BlueJ. Mit diesem Befehl kann man Java-Objekte inspizieren (untersuchen).

Wenn Sie ***Inspect*** aufrufen, erscheint der **Objektinspektor**, eine rote Dialogbox:



1.3 - 4 *Beim Inspizieren eines Objektes werden seine Attribute sichtbar*

Was man hier sieht, sind die **Attribute** der Klasse **Circle**, nämlich **diameter**, **xPosition**, **yPosition**, **color** und **isVisible**. Jedes dieser fünf Attribute hat einen **Attributwert** (weiße Kästchen). Das Attribut **diameter** (Durchmesser) hat einen Wert von `30` (Pixel), das Attribut **xPosition** einen Wert von `20` und so weiter.

Attribut	Datentyp	Bedeutung	Attributwert
diameter	`int`	Durchmesser	`30`
xPosition	`int`	Koordinaten des Kreis-Mittelpunktes	`20`
yPosition	`int`		`60`
color	`String`	Farbe	`"blue"`
isVisible	`boolean`	gibt an, ob das Objekt sichtbar ist	`false`

1.3 - 5 *Die fünf Attribute eines Circle-Objektes.*

Machen Sie sich den Unterschied zwischen **Attribut** und **Attributwert** klar:
Jedes **Circle**-Objekt hat ein Attribut **diameter**, weil das bei der Definition der Klasse **Circle** so festgelegt wurde. Zwei verschiedene **Circle**-Objekte können aber *unterschiedliche* Durchmesser haben. Ein großer Kreis hat vielleicht einen Durchmesser von 100 Pixeln, ein kleiner Kreis einen von 20 Pixeln. Die beiden **Circle**-Objekte haben dann unterschiedliche **Attributwerte**.

Versuchen wir nach diesen Überlegungen ein paar Definitionen:

Attribute

Attribute repräsentieren bestimmte **Eigenschaften** eines Objektes, zum Beispiel Größe, Position, Farbe etc.

Die Attribute werden bei der Definition der Klasse allgemein festgelegt, und jedes Objekt dieser Klasse besitzt alle Attribute.

Die **Attributwerte** können aber bei verschiedenen Objekten einer Klasse unterschiedlich sein.

Status bzw. Zustand

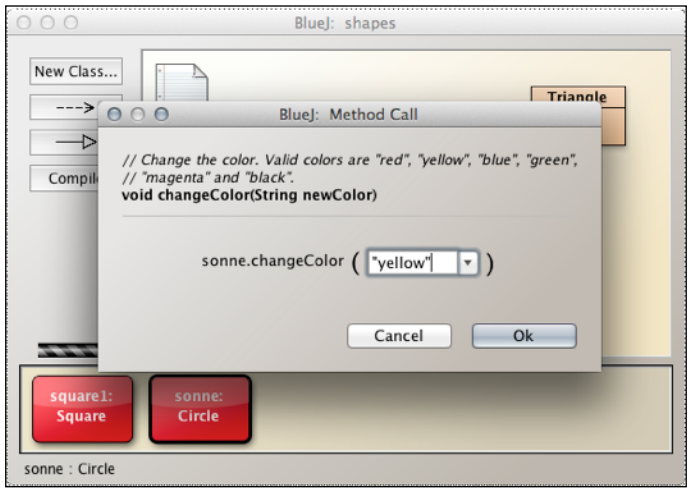
Als **Status** eines Objektes bezeichnet man die **Gesamtheit aller Attributwerte** eines Objektes.

Objekte der gleichen Klasse, bei denen *sämtliche* Attributwerte übereinstimmen, haben dann den *gleichen* Status.

Schritt 5 - manipulierende Methoden

Wie jedermann weiß, ist unsere Sonne groß und gelb und nicht klein und blau. Sie sollen jetzt aus dem "blauen Zwerg" eine normale gelbe Sonne machen.

➪Rufen Sie mit der rechten Maustaste die Methode **changeColor()** auf.



***1.3-6** komfortable Eingabe eines Parameters in eine BlueJ-Methode*

Es erscheint ein **Dialogfenster**, das Sie ausführlich über die Bedeutung der Methode aufklärt. In das **Textfeld** müssen Sie die Farbe des Kreises eintippen. Wichtig ist, dass Sie die Farbe in **Anführungszeichen** setzen.

➪Schreiben Sie den String (die Zeichenkette) `"yellow"` in das Textfeld und klicken Sie dann den Ok-Button.

Manipulierende Methode

Unter einer **manipulierenden Methode** versteht man eine Methode, mit der man den Wert eines **Attributes** (oder mehrerer Attribute) gezielt verändern kann.

Sie haben bisher zwei Methoden der Klasse **Circle** kennengelernt: **makeVisible()** und **changeColor()**. Beide Methoden sind manipulierende Methoden. Die Methode **makeVisible()** verändert ein Attribut namens **isVisible**. Vor dem Aufruf von **makeVisible()** hat dieses Attribut den Wert `FALSE`, das Objekt ist also nicht sichtbar. Nach dem Aufruf von **makeVisible()** hat das Attribut dagegen den Wert `TRUE`, also ist das Objekt sichtbar. Ähnlich ist es bei **changeColor()**. Vor dem Aufruf dieser Methode hat das Attribut **color** den Wert `"blue"`. Wenn wir die Methode **changeColor()** aufrufen und als neue Farbe `"yellow"` angeben, hat das Attribut **color** anschließend den Wert `"yellow"`.

Theorie - Parameter

Können Sie den Unterschied zwischen den beiden Methoden **makeVisible()** und **changeColor()** mit wenigen Worten zusammenfassen?

Die Methode **makeVisible()** kann *einfach so* aufgerufen werden. Es sind *keine weiteren Angaben* notwendig. Der Aufruf der Methode macht den vorher unsichtbaren Kreis sichtbar, indem sie den Wert des Attributes **isVisible** verändert.

Die Methode **changeColor()** kann *nicht einfach so* aufgerufen werden. Wenn Sie die Farbe eines Kreises ändern wollen, so müssen sie genau angeben, *welche Farbe* der Kreis haben soll. Die Methode benötigt eine *zusätzliche Information*. In der Informatik bezeichnet man solche zusätzlichen Informationen für Methoden als **Parameter**.

Parameter

Unter einem **Parameter** versteht man eine **Variable**, die einer Methode beim Aufruf übergeben wird. Die Methode kann diese Variable dann auswerten, beispielsweise um einen Attributwert gezielt zu verändern.

Variable

Unter einer **Variable** versteht man eine Stelle im Arbeitsspeicher des Rechners, die Daten speichern kann (zum Beispiel Zahlen oder Zeichenketten) und die mit einem definierten Namen angesprochen werden kann.

Die manipulierende Methode **changeColor()** benötigt genau *einen* Parameter, nämlich die neue Farbe. Es gibt aber auch Methoden, die *zwei*, *drei* oder *mehr* Parameter benötigen. Wenn Sie zum Beispiel ein Objekt der Klasse **Dreieck** anlegen und dann die Größe ändern wollen, so müssen Sie die neue Höhe und die neue Breite des Dreiecks eingeben, also zwei Parameter. Dies ist erforderlich, weil die manipulierende Methode zur Größenänderung zwei Attribute der Triangle-Objekte gleichzeitig verändert, nämlich **height** und **width**.

Übung 1.3-1

Erzeugen Sie zwei Objekte der Klasse **Triangle** und verändern Sie die Farbe, die Position und die Größe der beiden Dreiecke.

Schritt 6 - Datentypen

Sie können sich bestimmt schon denken, *wie* die Größe der Sonne verändert werden kann. Sie müssen die Methode **changeSize()** aufrufen. Auch diese Methode benötigt einen Parameter, nämlich die neue Größe, die der Kreis haben soll. Im Gegensatz zu **changeColor()** dürfen Sie jetzt keinen **String** (keine **Zeichenkette**) in das Textfeld des Dialogfensters eintippen, sondern es wird ein anderer **Datentyp** erwartet, nämlich ein **Zahlenwert**.

➪Tippen Sie jetzt die Zahl 100 ein - ohne Anführungszeichen wohlgemerkt.

Haben Sie es bemerkt? Es gibt in Java unterschiedliche **Typen** von Parametern. Bei **changeColor()** müssen Sie einen *Text* in Anführungszeichen eintippen - einen so genannten **String** (deutsch: **Zeichenkette**). Bei **changeSize()** dagegen müssen Sie eine *ganze Zahl* eingeben. Bei jedem Parameter muss man genau beachten, zu welchem **Datentyp** er gehört.

Datentypen

Der **Datentyp** legt den **Wertebereich** einer Variable, eines Parameters oder eines Attributs fest.

Jetzt hat der Kreis nicht nur eine andere Farbe, sondern ist auch größer geworden. Durch den Aufruf der Methode **changeColor()** haben Sie das **Attribut** **color** des Objektes **kreis1** verändert. Vorher hatte das Attribut **color** den **Attributwert** "blue", jetzt hat es den Attributwert "yellow". Durch den Aufruf der Methode **changeSize()** wurde entsprechend der Attributwert des Attributes **diameter** geändert.

Für Experten: Set-Methoden

In der Fachliteratur werden manipulierenden Methoden häufig auch als **Set-Methoden** bezeichnet (vom engl. *to set* = setzen, verändern).

Es ist Konvention mit einer solchen Set-Methode jeweils nur *ein einziges* Attribut zu verändern.

Weiter ist es Konvention, dass eine Set-Methode aus dem Präfix "set" besteht, dem der Name des Attributes angehängt wird. Eine Methode zum Ändern des Attributes **color** sollte also **setColor()** heißen und nicht **changeColor()**. Achten Sie auf die Konvention, dass man keine Unterstriche zur Trennung von Worten verwendet, sondern die Anfangsbuchstaben eines neuen Wortes groß schreibt, also nicht **set_color()**, sondern **setColor()**.

Im Deutschen schreibt man dann häufig "setze", eine Methode zum Ändern des Attributes **farbe** würde man dann also **setzeFarbe()** nennen.

Für Abiturienten: Aufträge

In der deutschen Informatik-Didaktik bezeichnet man manipulierende Methoden außerdem oft als "Aufträge". Siehe hier zum Beispiel "Materialien zu den zentralen Abiturprüfungen im Fach Informatik - Objektorientierter Ansatz" auf der Webseite des Schulministeriums NRW (http://www.standardsicherung.schulministerium.nrw.de/abitur/getfile.php?file=1706)

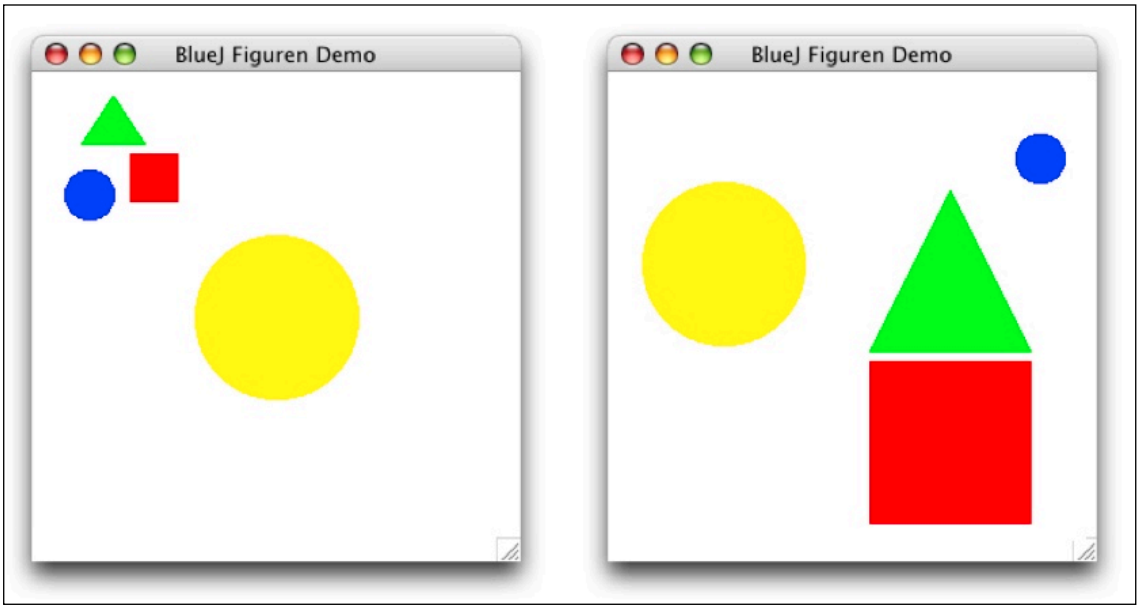
Übung 1.3-2

Erzeugen Sie in Ihrem BlueJ-Projekt folgende Objekte: Einen Kreis namens **sonne**, einen Kreis namens **mond**, ein Quadrat namens **haus** und ein Dreieck namens **dach**.

Rufen Sie dann für jedes der Objekte die Methode **makeVisible()** auf. Ihr "Gemälde" sollte dann ungefähr so aussehen wie in Abbildung 1-17, falls Sie den Kreis **sonne** bereits wie oben besprochen verändert haben. Wenn nicht, sehen Sie nur *einen* blauen Kreis, der in Wirklichkeit aber aus *zwei* Kreisen **sonne** und **mond** besteht, die jedoch die gleichen Koordinaten haben und daher übereinander liegen.

Versuchen Sie dann durch **geschickten Aufruf geeigneter Methoden** ein Bild zu malen, wie es in der Abbildung 1.3-7 rechts dargestellt ist.

Schreiben Sie sich genau auf, welche Methoden Sie in welcher Reihenfolge mit welchen Parametern aufgerufen haben.



***1.3-7** Die vier Objekte vor und nach der Übung*

Theorieteil

1.4 Klassen und Objekte (I1 / A, M, D)

Die Auto-Analogie

In der Garage Ihres Nachbarn steht ein weißer Passat. Die Farbe bröckelt schon etwas ab, weil er 12 Jahre alt ist. Vorne rechts in der Windschutzscheibe ist ein kleiner Riss, und die Antenne ist an zwei Stellen verbogen. Das Auto, das Sie jeden Tag sehen, ist ein *ganz bestimmter* Passat, den man anhand dieser Eigenschaften oder Attribute eindeutig von anderen Fahrzeugen des gleichen Typs unterscheiden kann.

Etwas völlig anderes ist es, wenn Sie von einem Bekannten gefragt werden: "Wie findest Du eigentlich den neuen Passat von VW?" Damit ist kein individuelles Fahrzeug mehr gemeint, sondern die ganze Baureihe.

Was hat das jetzt alles mit Klassen und Objekten zu tun? Nun, die *Baureihe Passat* entspricht einer **Klasse**. Der *Passat in der Garage* ist aber ein **Objekt**.

Eine Klasse ist etwas Abstraktes, man kann "die Baureihe Passat" nicht anfassen oder sehen. Wenn man dagegen einen bestimmten Passat sieht, so sieht man immer ein konkretes Objekt. Den "Passat des Nachbarn" kann man anfassen, man kann ihn fahren und so weiter.

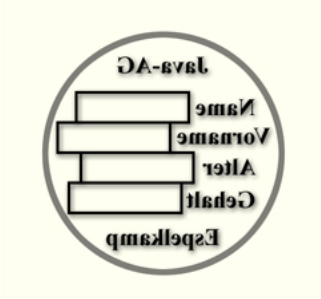
Eine Java-Klasse kann man dagegen durchaus sehen - hier hinkt der Vergleich also etwas. Wenn wir auf den Kasten der Klasse **Circle** klicken, öffnet sich ein Fenster, in dem der Java-Quelltext der Klasse zu sehen ist.

Von einer Klasse kann es viele Objekte geben: Von der Baureihe Passat fahren viele Tausend Exemplare in Deutschland herum. Jedes Passat-Objekt unterscheidet sich von anderen Passat-Objekten. Der eine Passat ist weiß, der andere blau. Ein Passat hat 90 PS, der andere 110. Einer hat einen Sylt-Aufkleber, der andere einen Heck-Spoiler. Jedes Passat-Objekt hat also bestimmte **Attribute** (Eigenschaften), anhand derer es man erkennen kann. Selbst wenn zwei Passate absolut gleich aussehen (gleiche Farbe, gleiche PS-Zahl, gleiche Beule vorne links, weil sie den gleichen Fußgänger erwischt haben), so haben sie trotzdem auch ein paar unterschiedliche Attribute: Die Fahrgestellnummer und die Kennzeichen sind beispielsweise verschieden. Selbst wenn es einer Bande von Autoschiebern gelingt, die Fahrgestellnummer und die Kennzeichen zu fälschen, so dass sie bei beiden Fahrzeugen übereinstimmen, befinden sich die beiden Autos doch an unterschiedlichen Koordinaten.

Dies alles gilt auch für Java-Klassen. Von der Klasse **Circle** kann man beliebig viele Objekte anlegen. Allerdings ist es in Java durchaus möglich, zwei absolut gleiche Objekte einer Klasse zu erzeugen, die in allen Attributen übereinstimmen. In einem Punkt würden sich diese beiden Objekte allerdings doch unterscheiden, nämlich in ihrem Namen. Man kann in einem Java-Programm nicht zwei Objekte anlegen, die den gleichen Namen haben.

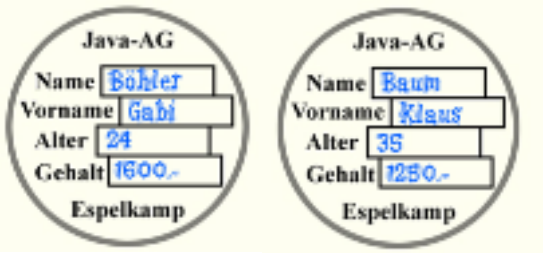
Die Attribute von Objekten sind *veränderbar*: Man kann die krumme Antenne eines Passats wieder geradebiegen oder die schadhafte Windschutzscheibe reparieren lassen. Und die Position eines Passat-Objektes ändert sich sowieso ständig, wenn das Auto gefahren wird. Hier ist die Analogie vollständig. Auch Java-Objekte kann man verändern, indem man die Attributwerte ändert. Zu diesem Zweck gibt es die manipulierenden Methoden.

Die Stempel-Analogie



1.4-1 *Ein Stempel (von unten)*

Betrachten Sie den Stempel eines Gehaltsbüros einer Firma (Abb. 1-19). Der Angestellte in diesem Büro haut diesen Stempel immer auf Karteikarten, und dann trägt er die vier Angaben (ich habe dieses Beispiel stark vereinfacht) auf die Karteikarte ein. Das könnte z.B. so aussehen wie in Abbildung 1-20.



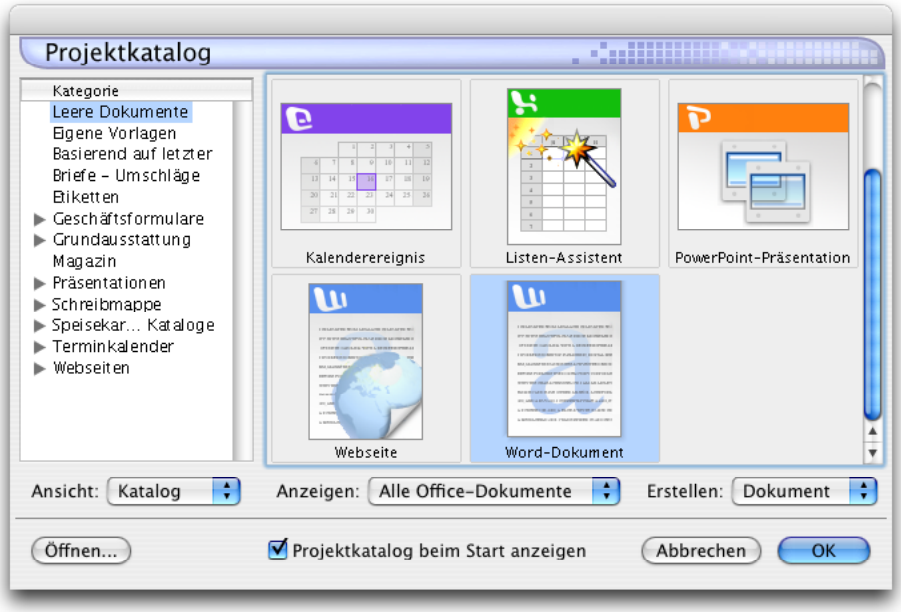
1.4-2 *Zwei Stempelabdrücke, mit Daten ausgefüllt*

Zunächst macht der Angestellte die Karteikarte von *Gabi Böhler* fertig. Als nächstes bearbeitet er die Karte von *Klaus Baum*. Diese Karteikarte unterscheidet sich von der ersten, obwohl sie mit dem gleichen Stempel hergestellt wurde. Es ist nicht schwer zu erraten, was hier die **Klasse** und was ein **Objekt** sein soll. Der Stempel ist die Klasse, während die ausgefüllten Stempelabdrücke auf den Karteikarten die Objekte sind. Bisher liegen zwei Objekte vor, die sich in ihren Attributwerten unterscheiden: Name, Vorname, Alter und Gehalt sind unterschiedlich. Wenn der Angestellte im Büro aber fleißig ist, wird er im Laufe des Tages noch viel mehr solcher Objekte produzieren. Und zwar mit seinem **Werkzeug zur Herstellung von Objekten**, dem Stempel. Ähnlich wie der Stempel ist auch eine Klasse eine Vorlage zur Herstellung von Objekten.

Dies ist eigentlich eine sehr schöne Analogie. Den Stempel muss man erst herstellen, damit man Abdrücke machen kann. Ähnlich muss man eine Java-Klasse zuerst programmieren, man muss den Quelltext aufschreiben und speichern und kompilieren. Erst dann kann man Objekte der Klasse anlegen.

Die Word-Analogie

Sie haben sicherlich schon einmal mit Word-Formatvorlagen gearbeitet. Wenn man zum Beispiel einen Geschäftsbrief schreiben will, so doppelklickt man einfach auf die gewünschte Formatvorlage. Word erzeugt dann ein neues Dokument, das genauso aufgebaut ist, wie in der Formatvorlage festgelegt wurde. Die Formatvorlage ist also eine **Vorschrift zur Konstruktion von Dokumenten**. Wenn Sie das Word-Dokument speichern wollen, müssen sie ihm einen eindeutigen Namen geben. Danach können Sie ein zweites Dokument von der gleichen Vorlage erzeugen (Abb. 1-21), ein drittes und so weiter.



***1.4-3** Formatvorlagen bei Word*

Der Zusammenhang mit Klassen und Objekten sollte Ihnen jetzt klar sein: Die Formatvorlage entspricht einer Klasse, und das konkrete Dokument ist ein Objekt. Auch hier gilt wieder: **Klassen sind Vorlagen zur Erstellung von Objekten**, und die verschiedenen Objekte einer Klasse unterscheiden sich voneinander durch unterschiedliche Attributwerte.

Klassen im Spiegel der Schulbücher

Der Begriff der **Klasse** wird in den verschiedenen Schulbüchern teils unterschiedlich definiert. Zunächst einmal möchte ich meine eigene, unvollkommene Definition zum Besten geben:

Klassen

Klassen sind Vorlagen zur Erstellung von **Objekten**.

Bei der Konstruktion einer Klasse werden die **Attribute** und die **Methoden** definiert, die allen Objekten dieser Klasse eigen sind. Die konkreten **Attributwerte** können sich jedoch von Objekt zu Objekt unterscheiden.

Nun wollen wir einmal verschiedene Definitionen aus unterschiedlichen Büchern vergleichen und die wesentlichen Aspekte aufzeigen.

Die erste Definition, die mir in die Hände kommt, ist aus dem Buch "Objektorientierte Softwareentwicklung mit UML" von Peter FORBRIG (München, Wien, 2002).

Klasse

Eine Klasse beschreibt eine Sammlung von Objekten mit gleichen Eigenschaften (Attributen), gemeinsamer Funktionalität (Methoden), gemeinsamen Beziehungen zu anderen Objekten und gemeinsamer Semantik.

Hier wird ebenfalls auf die gemeinsamen Attribute und Methoden eingegangen. Neu in dieser Definition ist die Erwähnung der gemeinsamen Beziehungen zu anderen Objekten. Dieses Thema habe ich in diesem Skript noch nicht behandelt, daher werde ich es hier auch nicht vertiefen.

In dem Informatik-Schulbuch des Duden-Verlags, "Objektorientierte Programmierung mit BlueJ" (Berlin 2008) habe ich folgende Definition des Begriffs Klasse gefunden:

Klasse

Eine **Klasse** beschreibt Objekte mit gleichen Attributen und gleichen Methoden. Eine Klasse kann mit einer Fabrik verglichen werden, sie kann viele Objekte erzeugen. Die Attribute eines **Objekts** nehmen bestimmte Werte an. Ein Objekt bietet **Dienste**, man sagt auch (öffentliche) **Methoden** an. Diese werden in der **Klassenkarte** notiert.

Eine interessante Definition, die hauptsächlich auf Objekte, weniger auf Klassen eingeht. Der Vergleich mit einer Fabrik ist nicht so gut, finde ich. Eine Fabrik kann durchaus viele unterschiedliche Objekte erzeugen, aus einer Klasse kann man aber immer nur Objekte des gleichen Typs herstellen. Außerdem gehört eine solche Analogie nicht in die *Definition* eines Begriffs, sondern in die *Erläuterung* der Definition.

Ungewöhnlich ist hier auch, dass die Methoden eines Objektes bzw. einer Klasse als **Dienste** bezeichnet werden. Mit dem Begriff **Klassenkarte** ist ein einfaches UML-Klassendiagramm gemeint.

Folge 2 - Wir programmieren

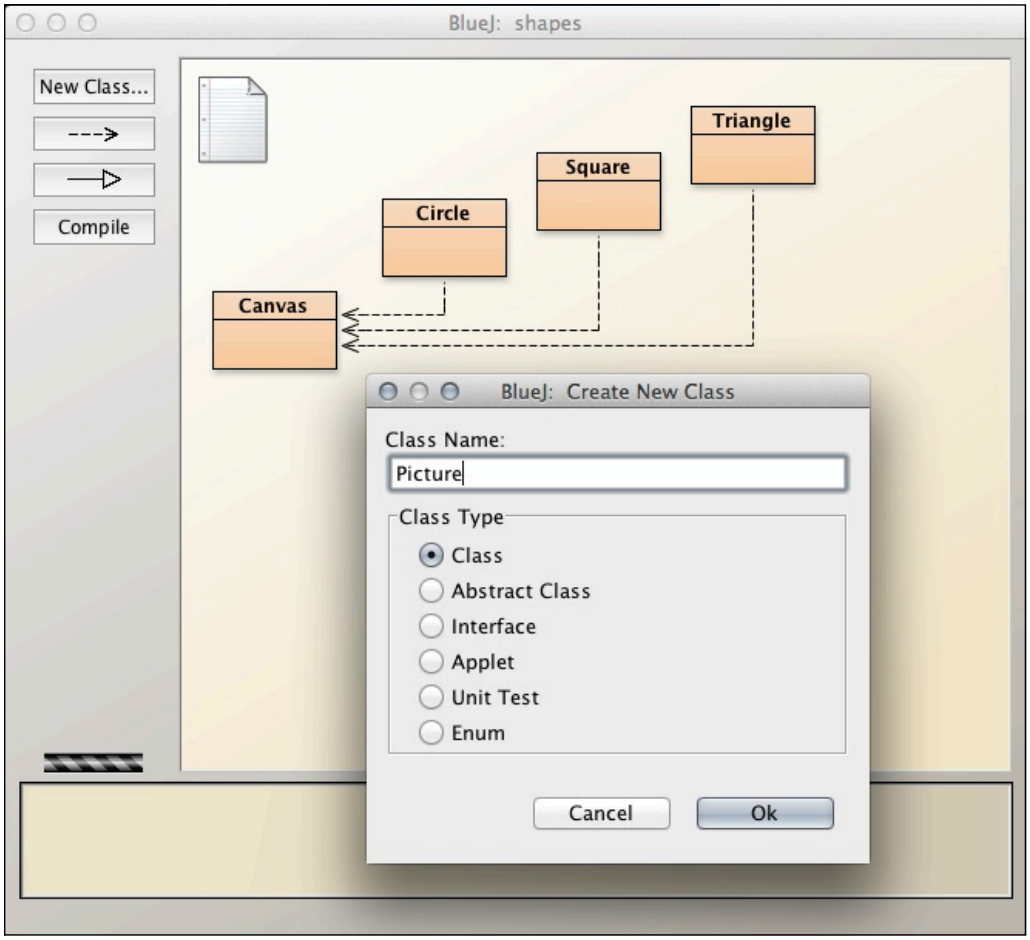
Unterrichtsvorhaben

2.1 Wir programmieren ein Bild (I1,I3 / M,I)

In den Übungen der Folge 1 haben Sie die Methoden der Klassen **Circle**, **Square** und **Triangle** *mithilfe der rechten Maustaste* aus einem Kontextmenü ausgewählt. Wenn Sie die Farbe des Kreis-Objektes **sonne** verändern wollten, mussten Sie erst die Methode **changeColor()** mit der Maus anwählen und dann in das Dialogfenster den Wert des Parameters eingeben: `"yellow"`. Ähnlich mussten Sie bei der Änderung der Größe, der Änderung der Position und so weiter verfahren. In dieser Folge werden Sie eine völlig andere Verfahrensweise kennen lernen, mit der Sie das Gleiche sehr viel eleganter erreichen können. Sie werden programmieren!

Schritt 1 - Neue Klasse erzeugen

➪Öffnen Sie das Projekt `"shapes"` und klicken Sie den Button zur Erzeugung einer neuen Klasse an. Sie werden aufgefordert, der Klasse einen ***Namen*** zu geben und einen ***Typ*** zuzuweisen. Wählen Sie als Name `"Picture"` und als Typ ***Class***, so wie in der folgenden Abbildung gezeigt.



2.1 - 1 *Erzeugen einer neuen Klasse*

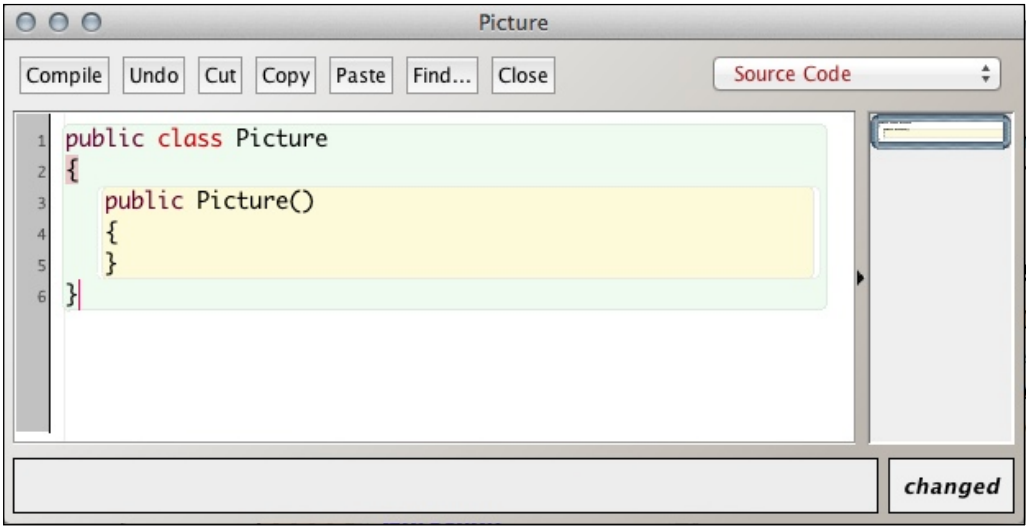
Schritt 2 - Quelltext eingeben

➪Doppelklicken Sie auf die Klasse **Picture**.

Es öffnet sich ein ***Editor***-Fenster mit dem **Quelltext** der Klasse. Die Programmierer von BlueJ haben Ihnen in gut gemeinter Absicht schon viel Arbeit abgenommen. Am besten lernen Sie Java jedoch, wenn Sie den von BlueJ erzeugten Quelltext *komplett löschen* und dann folgenden eigenen Quelltext in das Editor-Fenster eintippen:

```
public class Picture
{
   public Picture()
   {
   }
}
```

Die neueren Versionen von BlueJ stellen den eingegebenen Quelltext sehr übersichtlich dar - hier ein Bildschirmphoto des Editor-Fensters:



2.1 - 2 *Bildschirmphoto des Editor-Fensters*

Der äußere (hellblaue) Kasten repräsentiert die gesamte Klasse **Picture**, während der innere (hellgelbe) Kasten den **Konstruktor** der Klasse repräsentiert.

Konstruktor

Eine Konstruktor ist eine spezielle Methode einer Klasse, die immer dann aufgerufen wird, wenn mithilfe des `new`-Befehls ein *neues* Objekt dieser Klasse erzeugt wird.

Betrachten wir die Klasse **Circle**. Wenn Sie im BlueJ-Fenster mit der rechten Maustaste auf diese Klasse klicken und dann den Befehl ***new Circle()*** auswählen, müssen Sie zunächst einen Namen für das zu erzeugende Objekt eingeben, beispielsweise **sonne**. Wenn Sie dann mit dem ***Ok***-Button bestätigen, wird automatisch, ohne dass Sie es bemerken, von BlueJ der Konstruktor der Klasse **Circle** aufgerufen. Dieser Konstruktor sorgt dann dafür, dass ein Objekt der Klasse **Circle** mit der Bezeichnung "sonne" erzeugt wird. Außerdem sorgt der Konstruktor dafür, dass das Objekt eine bestimmte Ausgangsfarbe, Ausgangsgröße und Ausgangsposition hat.

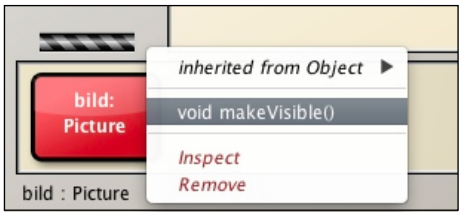
Schritt 3 - Methoden

Neben dem Konstruktor haben Klassen normalerweise weitere Methoden. Wir wollen die Klasse **Picture** jetzt um eine Methode namens **makeVisible()** ergänzen:

```
public class Picture
{
   public Picture()
   {
   }

   public void makeVisible()
   {
   }
}
```

Ich werde in Zukunft den neu hinzugekommenen Quelltext immer in der Farbe Blau schreiben, den alten, unveränderten Quelltext in der Farbe Schwarz.

➪Kompilieren Sie den Quelltext, erzeugen Sie ein Objekt **bild** der Klasse **Picture**, und klicken Sie dann mit der rechten Maustaste auf **bild**.



2.1 - 3 *Die neue Methode erscheint im Objektinspektor*

Tatsächlich taucht jetzt die neue Methode **makeVisible()** im Kontextmenü des Objektes **bild** auf. Allerdings passiert nichts, wenn Sie auf die Methode klicken. Das ist aber auch kein Wunder, schließlich haben Sie noch *keine Befehle* in die Methode hineingeschrieben.

Schritt 4 - Das Bild sichtbar machen

➪Ergänzen Sie bitte den Quelltext der Klasse **Picture** folgendermaßen:

```
public class Picture
{
   Circle sonne, mond;

   public Picture()
   {
      sonne = new Circle();
      mond  = new Circle();
   }

   public void makeVisible()
   {
       sonne.makeVisible();
       mond.makeVisible();
   }
}
```

Mit der Zeile

```
Circle sonne, mond;
```

haben wir die Klasse **Picture** mit zwei Objekten der Klasse **Circle** ausgestattet. Man sagt auch: **Die Klasse Picture hat zwei Objekte der Klasse Circle** und spricht von einer **HAT-Beziehung** zwischen den beiden Klassen.

Der Konstruktor der Klasse **Picture** sorgt nun dafür, dass die beiden **Circle**-Objekte auch tatsächlich erzeugt werden:

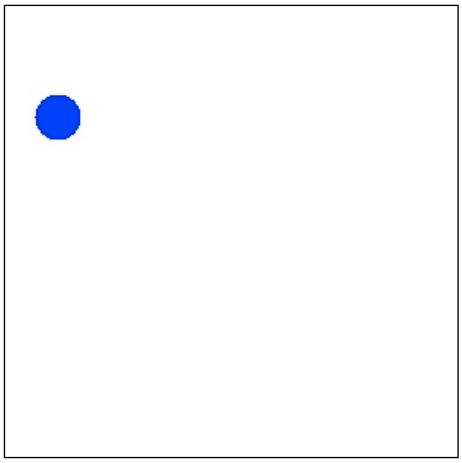
```
sonne = new Circle();
mond  = new Circle();
```

Im letzten Unterrichtsvorhaben haben Sie die **Circle**-Objekte noch erzeugt, indem Sie mit der rechten Maustaste auf den "Klassenkasten" für **Circle** geklickt und dann den Befehl ***new Circle()*** ausgewählt haben. Jetzt programmieren Sie jedoch richtig. Das heißt, Sie müssen diesen Befehl in den Quelltext des Konstruktors eintippen. Einmal für die Erzeugung des **Circle**-Objektes **sonne**, und dann noch einmal für die Erzeugung des **Circle**-Objektes **mond**.

Die neue Methode **makeVisible()** bekommt jetzt auch Arbeit. Sie ruft nämlich die **makeVisible()**-Methoden der beiden Objekte auf:

```
sonne.makeVisible();
mond.makeVisible();
```

Durch diesen doppelten Aufruf - für jedes der beiden Objekte einmal - werden die Objekte auf der Zeichenfläche sichtbar gemacht. Leider verläuft diese Sichtbarmachung nicht ganz so wie geplant. Man sieht nicht zwei Kreise, einen großen gelben und einen kleinen blauen, sondern nur einen Kreis, nämlich einen kleinen blauen.

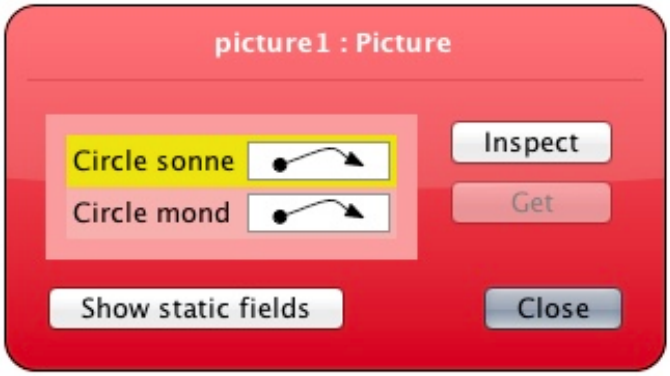


2.1 - 4 Man sieht leider nur einen Kreis

Wir wollen jetzt einmal untersuchen, woran das wohl liegen könnte.

Schritt 5 - Fehlersuche

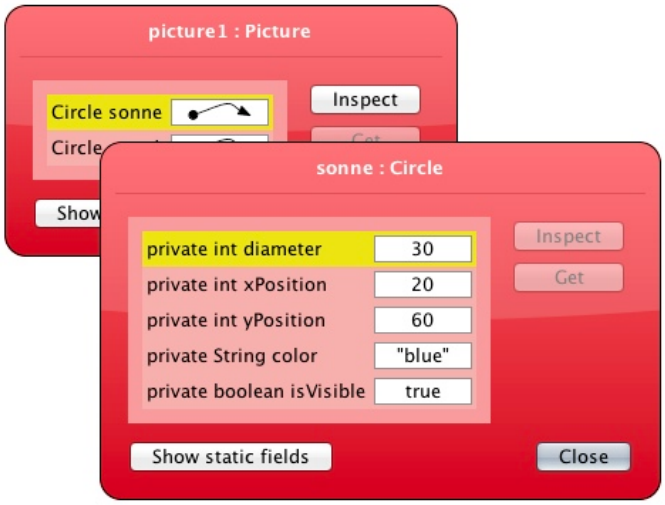
➪Erzeugen Sie ein Objekt **picture1** der Klasse **Picture** und inspizieren Sie das Objekt.



2.1 - 5 Die Attribute des Objektes picture1

Sie sehen hier die beiden Attribute **sonne** und **mond**, beides sind Objekte der Klasse **Circle**. Aber eigenartig - in den weißen Kästchen stehen gar keine Attributwerte! Oder doch? Was bedeuten die beiden Pfeile?

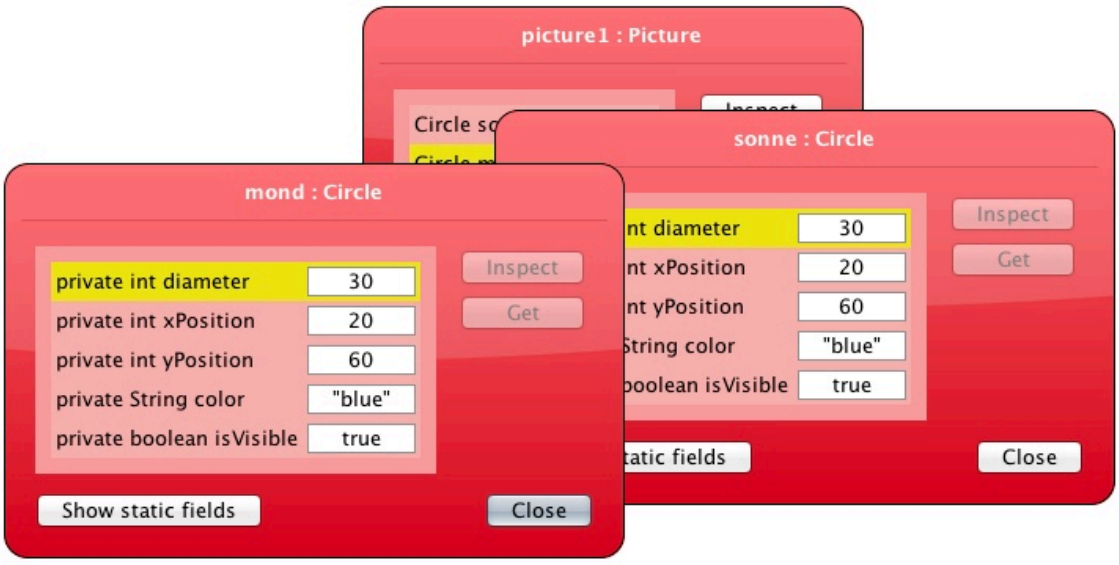
➪Führen Sie einen Doppelklick auf den oberen Pfeil aus.



***2.1 - 6** Die Attribute des Objektes sonne*

Es erscheint eine Box mit den Attributen des Objektes **sonne**. Hier sehen wir die Attributwerte.

➪Führen Sie entsprechend einen Doppelklick auf den unteren Pfeil - für das Objekt **mond** - aus. Vergleichen Sie dann die Attributwerte der beiden Objekte.



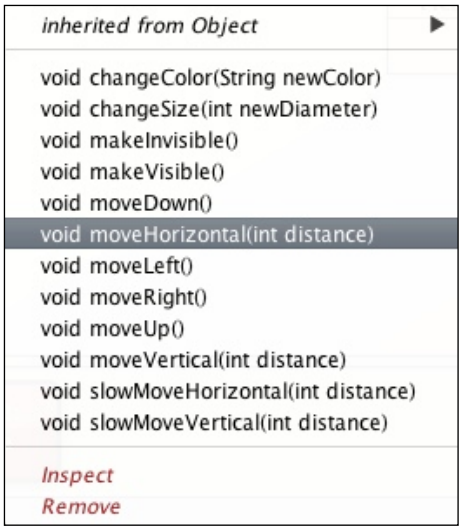
***2.1 - 7** Die Attributwerte von mond und sonne im Vergleich*

Inzwischen haben Sie es sicherlich gemerkt - die beiden Objekte haben absolut identische Attributwerte: gleichen Durchmesser, gleiche Position, gleiche Farbe, und beide sind sichtbar. Man könnte also sagen, die beiden Objekte haben exakt den gleichen **Status** - denn der Status ist ja nichts anderes als die Gesamtheit aller Attributwerte eines Objektes.

Auf der Zeichenfläche ist also nicht *ein* Kreis zu sehen, sondern man sieht *beide* Kreise. Nur liegen sie genau übereinander, daher sehen die Kreise aus wie ein Kreis.

Schritt 6 - Einen der beiden Kreise bewegen

Als Sie noch nicht programmierten, haben Sie in Folge 1 ein Kreisobjekt bewegt, indem Sie aus dem Kontextmenü eine geeignete Methode aufgerufen haben.



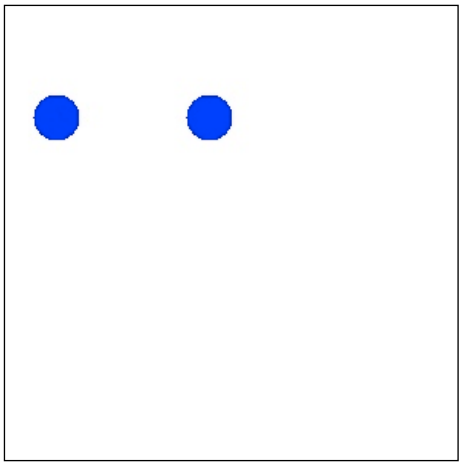
***2.1 - 8** Kontextmenü eines Circle-Objektes*

Die Methode **moveHorizontal()** würde sich zum Beispiel gut zum horizontalen Bewegen des Kreises um eine bestimmte Pixelzahl eignen, die als Parameter übergeben wird.

➪Bauen Sie einen solchen Befehl in den Quelltext der Klasse **Picture** ein:

```
public void makeVisible()
{
    sonne.makeVisible();
    mond.makeVisible();
    mond.moveHorizontal(100);
}
```

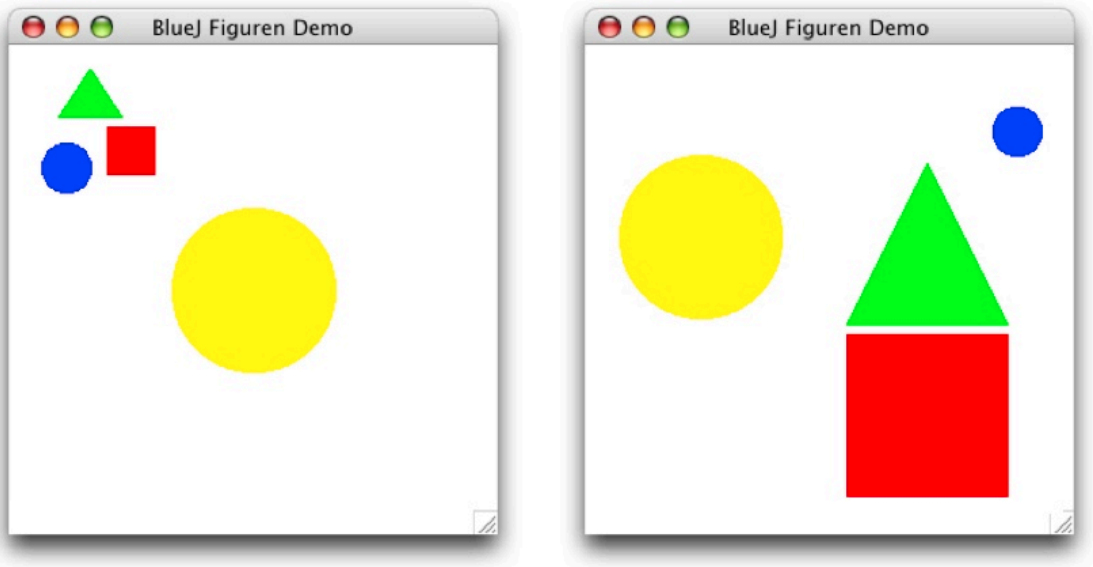
➪Kompilieren Sie die Klasse, erzeugen Sie ein Objekt **picture1** und führen Sie dann den **makeVisible()**-Befehl aus. Sie haben es geschafft - man sieht jetzt zwei kleine blaue Kreise.



2.1 - 9 *Man sieht jetzt die beiden Kreise*

Übung 2.1-1

1. Deklarieren Sie in die Klasse **Picture** folgende Objekte: Einen Kreis namens **sonne**, einen Kreis namens **mond**, ein Quadrat namens **haus** und ein Dreieck namens **dach**.
2. Erzeugen Sie die vier Objekte im Konstruktor der Klasse **Picture** mit dem **new**-Befehl.
3. Erweitern Sie die Methode **makeVisible()** der Klasse **Picture** so, dass die vier Objekte sichtbar werden. Allerdings stimmen weder die Positionen noch die Größen noch die Farben der Objekte mit dem Bild überein, das Sie zeichnen sollen.
4. Schreiben Sie eine Methode **draw()** für die Klasse **Picture**, welche dann die Hauptarbeit leistet, nämlich das Zeichnen des Bildes: Ein Haus mit einem Dach und einem Mond und einer Sonne am Himmel, ähnlich wie in der Folge 1.



2.1 - 10 *Das rechte Bild soll in Übung 2.1-1 von der draw()-Methode der Klasse Picture gezeichnet werden*

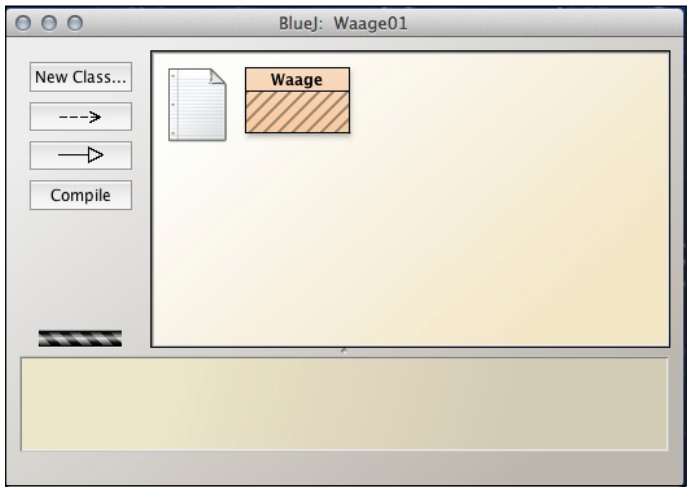
Folge 3 - Sie werden gewogen!

Unterrichtsvorhaben

3.1 Entwurf der Klasse "Waage" (I1,I2,I3 / M,I,D)

Schritt 1 - neues Projekt erstellen

➪Erzeugen Sie eine neues leeres Projekt mit der Klasse **Waage**:



***3.1 - 1** Das neue Projekt Waage*

➪Doppelklicken Sie auf die Klasse **Waage** und öffnen Sie so den Quelltexteditor. Entfernen Sie den kompletten Text und tippen Sie dann den Minimal-Quelltext der Klasse **Waage** ein:

```
public class Waage
{
   public Waage()
   {
   }
}
```

➪Kompilieren Sie den Quelltext, legen Sie ein Objekt **waage1** der Klasse **Waage** an und doppelklicken Sie auf den roten Kasten, der im BlueJ-Fenster dieses Objekt repräsentiert.

Der Objektinspektor zeigt noch nicht viel an. Aber wir haben ja auch noch kein einziges Attribut und keine einzige Methode implementiert. Und der Konstruktor ist noch leer, was also sollte der Objektinspektor auch schon anzeigen?

Schritt 2 - Grundlegende Methoden der Waage

Zunächst überlegen wir uns, was die Waage alles *können* soll - damit wären wir bei den **Methoden** der Klasse **Waage**, die das **Verhalten** der Objekte definieren.

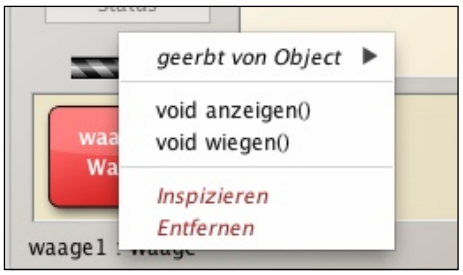
Das Verhalten einer normalen Waage sieht so aus: Sie steht auf dem Fußboden eines Raumes, normalerweise im Badezimmer. Irgendwann, meistens morgens vor dem Essen, stellt sich jemand auf die Waage, die dann das Gewicht der Person anzeigt, entweder analog (Zeiger), oder digital. Damit hätten wir bereits die ersten beiden Methoden festgelegt: **wiegen()** und **anzeigen()**.

➪Ergänzen Sie jetzt Ihren Quelltext folgendermaßen:

```
public class Waage
{
   public Waage()
   {
   }

   public void wiegen()
   {
   }

   public void anzeigen()
   {
   }
}
```

Die beiden neuen Methoden sind im Quelltext blau hervorgehoben.

➪Übersetzen Sie den Quelltext, erzeugen Sie ein Objekt **waage1** und klicken Sie mit der rechten Maustaste auf das Objekt.



***3.1-2** Die Methoden sind bereits zu sehen*

Man kann die beiden Methoden schon sehen. Aber wenn man versucht, die Methoden auszuführen, passiert nichts. Kein Wunder, denn die beiden Methoden enthalten noch keinen Java-Quelltext.

Schritt 3 - Attribute der Waage

Die Methode **wiegen()** soll das Gewicht der Person ermitteln, die sich auf die Waage stellt. Dieses Gewicht (in Kilogramm) muss in den Objekten der Klasse **Waage** *gespeichert* werden, damit es später von der Methode **anzeigen()** in der Textkonsole ausgegeben werden kann.

Für das *Speichern von Daten* sind allgemein Variablen zuständig. Variablen sind Bereiche im Arbeitsspeicher eines Rechners, die man sich wie kleine Schubladen in einem Schrank vorstellen kann. Man unterscheidet zwischen lokalen und globalen Variablen. Lokale Variablen sind Variablen, die nur innerhalb einer Methode verfügbar sind. Auf globale Variablen können dagegen alle Methoden eines Objektes zugreifen. In den meisten modernen objektorientierten Programmiersprachen werden solche globalen Variablen als **Attribute** bezeichnet. Das Attribut, welches das Gewicht der Person speichern soll, die sich gerade gewogen hat, wollen wir einfach als **gewicht** bezeichnen.

➪Ergänzen Sie Ihren Quelltext bitte wie folgt, kompilieren sie ihn und erzeugen Sie ein Objekt der Klasse Waage.

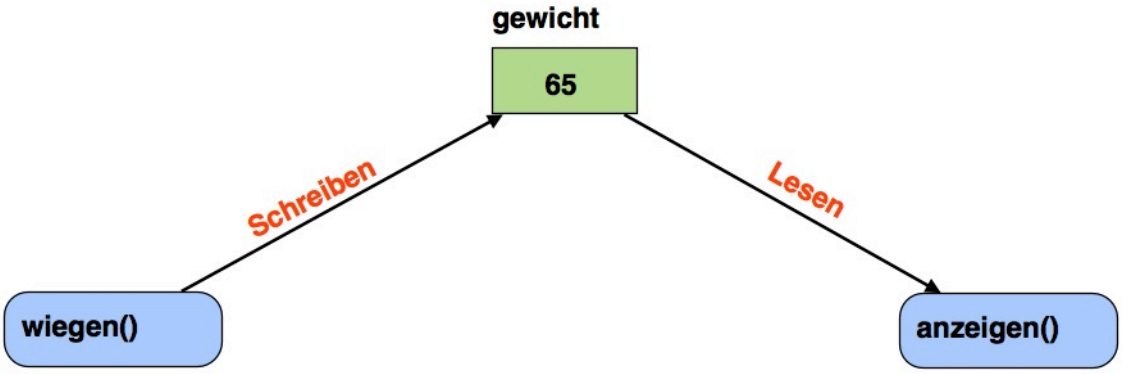
```
public class Waage
{
   int gewicht;

   public Waage()
   {
   }
   ...
```

Wenn man auf das Waage-Objekt doppelklickt, sieht man das Attribut **gewicht** im Objektinspektor. Der **Datentyp** dieses Attributes ist `int`, also eine ganze Zahl.

***3.1 - 3** Das definierte Attribut ist im Objektinspektor zu sehen*

Wie man gut sehen kann, hat das Attribut bereits einen Wert, nämlich `0`. Solche von Java vorgegebenen Werte bezeichnet man als **default-Werte**. Alle Zahlen-Attribute in Java haben den default-Wert 0. Alle String-Attribute (Zeichenketten) haben den default-Wert `""` (das ist ein leerer String; die beiden Anführungszeichen stehen direkt hintereinander, was man in diesem Skript nicht unbedingt gut sehen kann).

Vertiefung - Zusammenarbeit von Methoden und Attributen



3.1 - 4 Zusammenarbeit der Methoden und Attribute

In diesem Bild sehen, wie die beiden Methoden **wiegen()** und **anzeigen()** zusammenarbeiten sollen, wenn das Programm fertig ist, und welche Rolle dabei das Attribut **gewicht** spielt.

Die Methode **wiegen()** ermittelt das Gewicht der Person.

Die Methode **anzeigen()** zeigt das Gewicht der Person an.

Das Attribut **gewicht**, eine **globale Variable**, speichert das von der Methode **wiegen()** ermittelte Gewicht, so dass die Methode **anzeigen()** darauf zugreifen kann. Das Verändern / Manipulieren eines Attributwertes bezeichnet man in der Informatik auch als **Schreiben**, und das Ermitteln / Auswerten eines Attributwertes nennt man **Lesen**. Beim Lesen wird der Attributwert nicht verändert!

Methoden wie **wiegen()**, die einen Attributwert verändern, heißen übrigens verändernde oder **manipulierende Methoden**, während Methoden wie **anzeigen()**, die einen Attributwert nur lesen, **sondierende Methoden** heißen.

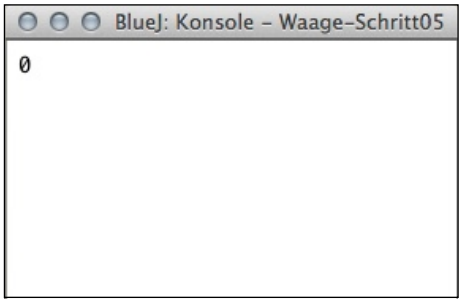
Schritt 4 - Die anzeigen()-Methode

Wir wollen jetzt die **anzeigen()**-Methode so ergänzen, dass der Wert des Attributs **gewicht** tatsächlich in der Textkonsole ausgegeben wird.

➪Ergänzen Sie den Quelltext der Methode **anzeigen()** wie folgt:

```
public void anzeigen()
{
    System.out.println(gewicht);
}
```

➪Kompilieren Sie den Quelltext, erzeugen Sie ein Objekt der Klasse und führen Sie die Methode **anzeigen()** mit der rechten Maustaste aus.



***3.1 - 5** Das Konsolenfenster mit der Ausgabe*

Es erscheint ein neues Fenster auf dem Bildschirm, das **Konsolenfenster** bzw. die **Konsole**. In diesem Fenster ist eine einzige Zahl links oben zu erkennen, nämlich die Null. Das ist genau der **default-Wert** des Attributes **gewicht**.

Exkurs - Die Konsole (I5)

Der Befehl `System.out.println()` dient allgemein zur Ausgabe von Zeichenketten, Zahlen etc. Die Ausgabe erfolgt in dem so genannten **Terminal Window**; auf Deutsch **Terminalfenster** oder **Konsole**.

***3.1 - 6** Ein Terminal, Anfang der 80er Jahre (aus einer alten Computer-Werbung).*

Unter diesem Terminalfenster müssen Sie sich so etwas Ähnliches vorstellen wie einen uralten Textcomputer, der die Zeichen hellgrün oder bernsteinfarben (das galt damals als besonders augenfreundlich) auf schwarzem Hintergrund darstellte.

Der alte Tandy TRS-80, ein um 1982 gebauter Desktop-Rechner für über 3000 DM (ohne Diskettenlaufwerke, geschweige denn Festplatte; die war noch gar nicht erfunden) zeigte auf seinem Bildschirm 16 Textzeilen mit jeweils 64 Buchstaben an. Graphiken konnten nur mithilfe besonderer Buchstabenzeichen dargestellt werden.

Schritt 5 - Die wiegen()-Methode

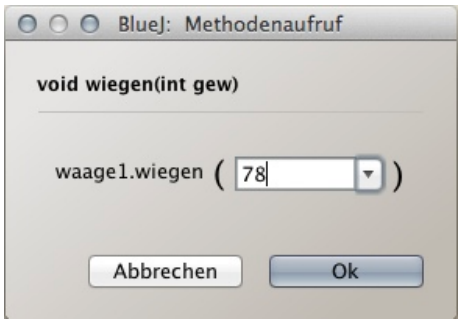
Wir wollen die Methode **wiegen()** nun so ergänzen, dass der Benutzer des Programms tatsächlich sein Gewicht in kg eingeben kann.

➪Ergänzen Sie den Quelltext der Methode **wiegen()** wie folgt:

```
public void wiegen(int gew)
{
    gewicht = gew;
}
```

➪Kompilieren Sie den Quelltext, erzeugen Sie ein Objekt der Klasse und führen Sie die Methode **wiegen()** mit der rechten Maustaste aus.

Es erscheint ein **Dialogfenster**, in dem die Methode genau beschrieben ist und das Sie gleichzeitig auffordert, eine Zahl einzutippen:



***3.1 - 7** Das Dialogfenster der Methode wiegen() fordert zum Eingeben einer Zahl auf*

➪Tippen Sie eine ganze Zahl, zum Beispiel 78, in das Textfeld des Dialogfensters ein und klicken Sie anschließend auf den ***Ok-Button***. Inspizieren Sie dann das Objekt mit einem Doppelklick.

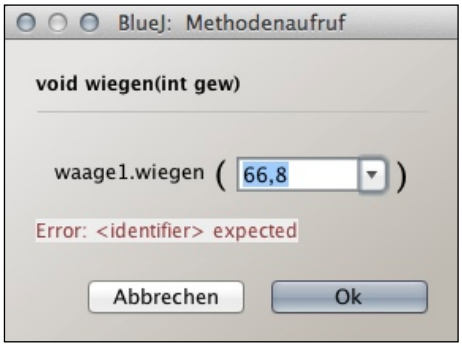
***3.1 -8** Der Wert des Attributes wird korrekt angezeigt*

➪Rufen Sie die Methode **anzeigen()** mit der rechten Maustaste auf.

Jetzt erscheint in der Konsole der zuvor eingegebene Wert, in unserem Beispiel also die 78.

Schritt 6 - Reelle Zahlen

Was machen Sie eigentlich, wenn Sie zufällig 66,8 kg wiegen. Wenn Sie diese Zahl in die Editbox des Dialogfeldes eingeben, bekommen Sie eine Fehlermeldung:

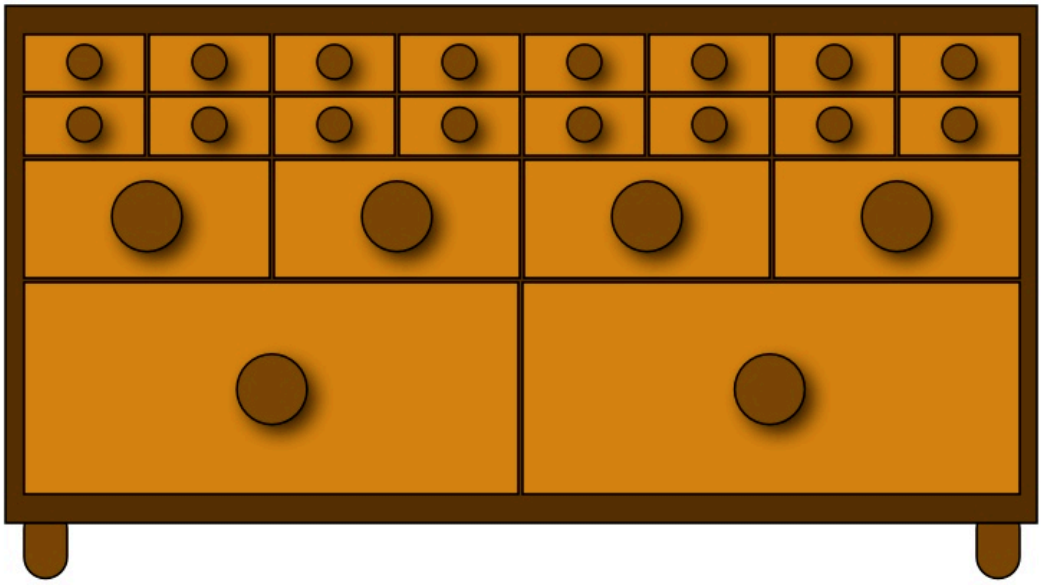


3.1 - 9 *Eingabe einer reellen Zahl führt zu einer Fehlermeldung*

Theorie - Datentypen

Um diese Fehlermeldung zu verstehen, müssen wir uns etwas mit dem Thema "Datentypen" beschäftigen.

Wenn Sie in Java eine Variable deklarieren, zum Beispiel das Attribut **gewicht**, dann müssen Sie sich die Variable so ähnlich vorstellen wie eine Schublade in einem Schrank. Dabei steht der Schrank für den Arbeitsspeicher des Computers, auf dem das Java-Programm läuft.



3.1 - 10 *Ein Schubladenschrank mit unterschiedlich großen Schubladen*

Die einzelnen Schubladen stehen für die Speicherplätze in diesem Arbeitsspeicher. Nun kann ein solcher Schrank unterschiedlich große Schubladen haben. Ähnlich verhält es sich auch mit dem Arbeitsspeicher eines Rechners, obwohl der Vergleich mit dem Schubladenschrank etwas hinkt. Wenn wir eine Variable deklarieren, beispielsweise **gewicht**, so müssen wir dem Computer mitteilen, wie

groß der benötigte Speicherplatz genau sein soll. Es ist nämlich ein Unterschied, ob wir eine *ganze* Zahl speichern wollen (int) oder eine *reelle* Zahl (double). Eine reelle Zahl benötigt mehr Speicherplatz als eine ganze Zahl, weil auch die Nachkommastellen gespeichert werden müssen.

Die Größe eines Speicherplatzes in einem Computer wird normalerweise in **Bit** gemessen. Ein Bit ist die kleinste Informationsmenge, die man sich überhaupt vorstellen kann. In einem einzigen Bit kann man eigentlich nur zwei Zustände speichern, nämlich "Ja" oder "Nein" bzw. "True" oder "False" bzw. "0" oder "1". Das liegt an den technischen Schaltkreisen, die sich in einem Computer befinden. Diese kleinsten Schaltkreise sind entweder "an" (aktiv) oder "aus" (passiv). Zwischenzustände gibt es hier nicht.

Bit bzw. bit

Maßeinheit für den Informationsgehalt, Abkürzung für binary digit, Binärziffer.

Schreibweise:

Der Name der Maßeinheit ist "Bit" und entspricht damit Bezeichnungen wie "Meter" oder "Kilogramm".

Das *Zeichen* für die Maßeinheit ist "bit", ähnlich wie "m" das Zeichen für die Maßeinheit "Meter" ist oder "kg" das Zeichen für die Maßeinheit "Kilogramm".

Üblicherweise werden 8 Bit zu einem **Byte** zusammengefasst. In einem Byte, das aus acht Nullen oder Einsen besteht, kann man ganze Zahlen zwischen 0 (00000000) und 255 (11111111) speichern. Meistens braucht man beim Programmieren aber größere Zahlen. Dazu eignen sich die verschiedenen Datentypen, die Java zur Verfügung stellt. Hier eine Auswahl:

Datentypen für Zahlen

Durch Angabe des Datentypen `int`, `float` oder `double` wird festgelegt, welchen **Wertebereich** die Zahlen haben können. Für die Programmiersprache Java gilt:

Datentyp	Größe		Wertebereich
int	32 bit	4 byte	-2.147.483.648 ... 2.147.483.647
long	64 bit	8 byte	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
float	32 bit	4 byte	$+/-1{,}4 \cdot 10^{-45}$ ... $+/-3{,}4 \cdot 10^{+38}$
double	64 bit	8 byte	$+/-4{,}9 \cdot 10^{-324}$ ... $+/-1{,}7 \cdot 10^{+308}$

Es gibt also zwei Datentypen für ganze Zahlen, nämlich `int` und `long`. Wenn man eine Variable als `int` definiert, kann man Zahlen mit einer Wortbreite von 32 bit bzw. 4 byte speichern, das sind 32

Nullen oder Einsen. Das erste Bit kennzeichnet dabei das Vorzeichen, so dass auf diese Weise auch negative ganze Zahlen untergebracht werden können. Für den eigentlichen Zahlenwert stehen dann allerdings nur noch 31 Stellen zur Verfügung. Falls ganze Zahlen größer als 2.1 Milliarden oder kleiner als -2.1 Milliarden benötigt werden, kann man die Wortbreite auf 64 bit verdoppeln. Dazu muss man die Variable dann als `long` deklarieren.

Für reelle Zahlen (Kommazahlen) stehen ebenfalls zwei Datentypen zur Verfügung, nämlich `float` und `double`. Eine float-Zahl wird in Java folgendermaßen verwaltet: Die erste Stelle steht für das Vorzeichen. Die nächsten 8 Stellen speichern den Exponenten, und die letzten 23 Stellen die Mantisse, also den eigentlichen Zahlenwert. Bei double-Zahlen ist der Exponent 11 bit lang und die Mantisse 52 bit, double-Zahlen sind also wesentlich genauer als float-Zahlen. Standardmäßig verwendet Java für reelle Zahlen den Datentyp double; will man eine float-Zahl haben, so muss man dies explizit angeben. Einzelheiten dazu finden Sie in dem hervorragenden Buch "Java ist auch nur eine Insel" vom Galileo-Verlag (Abschnitt 12.2.3).

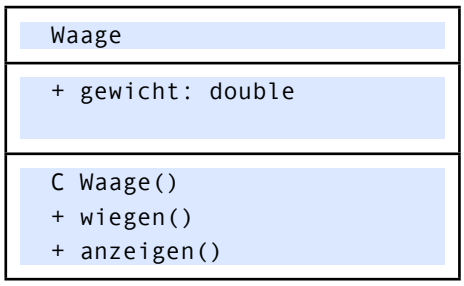
Schritt 6, Fortsetzung

➪Verändern Sie nun den Datentyp des Attributs **gewicht** sowie den Datentyp des Parameters `gew` von `int` zu `double`:

```
public class Waage
{
   double gewicht;

   public Waage()
   {
   }

   public void wiegen(double gew)
   {
       gewicht = gew;
   }
   ...
```

Theorie - Klassendiagramme

Betrachten Sie nun folgendes Bild:



***3.1 - 11** UML-Klassendiagramm der Klasse Waage*

Dies ist ein **UML-Diagramm** für die Klasse Waage, also ein sogenanntes **Klassendiagramm**. Das **Klassendiagramm** besteht aus einem Kasten, der in drei Abschnitte unterteilt ist.

In dem obersten Abschnitt steht einfach der **Name** der Klasse, hier also "Waage".

Der zweite Abschnitt enthält die **Attribute** der Klasse. Da unsere Klasse **Waage** bisher nur das Attribut **gewicht** besitzt, enthält dieser Abschnitt des Klassendiagramms auch nur diesen einen Eintrag. Achten Sie auf die **Syntax** im UML-Diagramm. Im Gegensatz zur Programmiersprache Java wird im UML-Diagramm zuerst der Bezeichner des Attributs aufgeführt, dann kommt ein Doppelpunkt, und dahinter steht dann der Datentyp. Ein Semikolon am Ende der Zeile ist nicht nötig. Das kleine Plus-Zeichen vor dem Attribut heißt so viel wie: Das Attribut ist von außen zugänglich, es ist "öffentlich" oder "public". Andere Klassen können auf dieses Attribut zugreifen und es dabei auch verändern.

Der dritte Abschnitt des Klassendiagramms enthält die **Methoden** der Klasse. Zunächst kommt immer der Konstruktor, der durch ein vorangestelltes `C` gekennzeichnet wird. Dann kommen die einzelnen Methoden. Dabei unterscheidet man öffentliche (public) und nicht-öffentliche (private) Methoden. Ein `+` oder ein `-` gibt an, ob eine Methode `public` oder `private` ist.

Bei größeren Klassen, die 10, 15 oder mehr Attribute und 20, 30 oder mehr Methoden haben, werden im UML-Klassendiagramm nur die wichtigen Attribute und Methoden berücksichtigt. Welche Attribute und Methoden jeweils wichtig sind und welche nicht, hängt von dem Kontext ab, in dem das Klassendiagramm verwendet wird. Will man beispielsweise zeigen, auf welche Weise die vielen Daten in die Objekte eingelesen werden, so berücksichtigt man nur die Methoden, die für das Einlesen der Daten notwendig sind. Möchte man herausstellen, wie andere Klassen auf die Methoden eines Objektes zugreifen können, so wird man die privaten Attribute und Methoden nicht im Klassendiagramm berücksichtigen, weil andere Klassen grundsätzlich keinen Zugriff auf private Attribute und Methoden haben.

Schritt 7 - die Methode messen()

Wenn Sie verstanden haben, wie die Methode **wiegen()** funktioniert, sollten Sie jetzt keine Schwierigkeiten haben, eine ähnliche Methode für den Messvorgang zu programmieren.

Übung 3.1-1 (2 Punkte)

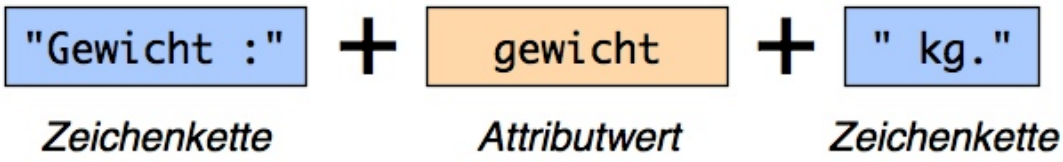
Ergänzen Sie den Quelltext der Klasse **Waage** um ein Attribut **groesse** und eine Methode **messen()**, die ähnlich arbeitet wie die Methode **wiegen()**, allerdings soll jetzt die **Körpergröße** der Person in cm gelesen und gespeichert werden. Erweitern Sie die Methode **anzeigen()** entsprechend.

Schritt 8 - eine bessere Ausgabe

➪Verändern Sie die Methode **anzeigen()** folgendermaßen und testen Sie das veränderte Programm:

```
public void anzeigen()
{
   System.out.println("Gewicht     : " + gewicht + " kg.");
}
```

Achten Sie dabei genau auf die Position der Anführungszeichen und der Plus-Symbole.
Das Plus-Symbol im **println()**-Befehl dient nicht zum *Addieren*, sondern zum *Zusammenfügen* der beiden Ausdrücke, zum **Konkatenieren**.



***3.1 - 12** Drei Ausdrücke werden konkateniert*

Hier kommt zuerst der **String** (**Zeichenkette**) `"Gewicht : "`. Der zweite Ausdruck ist das Attribut **gewicht**. Der dritte Ausdruck ist wieder ein String, nämlich `" kg."`. Die Leerzeichen in den Strings sorgen für ein ordentliches Aussehen der Meldung.

Übung 3.1-2 (2 Punkte)

Verändern Sie die **anzeigen()**-Methode so, dass folgender Text in der Konsole erscheint, falls Sie für Gewicht 70.3 kg und für Größe 180.6 cm eingegeben haben:

```
"Sie wiegen 70.3 kg und sind 180.6 cm groß!"
```

Schritt 9 - Berechnung des Idealgewichts

Die Formel zur Berechnung des Idealgewichts aus der Körpergröße lautet:

```
Idealgewicht = (Körpergröße - 100) * 0,9
```

➪Eine Methode **getIdealgewicht()**, die diese Aufgabe übernimmt, sieht so aus - ergänzen Sie bitte den Quelltext der Klasse **Waage** entsprechend:

```
public double getIdealgewicht()
{
   return (groesse-100) * 0.9;
}
```

Theorie - Sondierende Methoden

Wir haben hier gerade eine **sondierende Methode** geschrieben, die das Idealgewicht berechnet.

Sondierende Methode

Eine Methode, die den Wert eines Attributes ausliest und direkt zurück gibt (**get-Methode**), oder eine Methode, die aus Attributwerten ein Ergebnis berechnet und dies zurück gibt.

Sondierende Methoden des ersten Typs werden auch als **get-Methoden** bezeichnet. Hier ein typisches Beispiel:

```
public double getGroesse()
{
   return groesse;
}
```

Sondierende Methode des zweiten Typs lesen ebenfalls Attributwerte aus, geben aber nicht direkt diese Werte zurück, sondern berechnen aus den Werten ein Ergebnis, welches dann zurück geliefert wird. Die Methode **getIdealgewicht()** gehört zu diesem Typ sondierender Methoden. Auch die folgende **anzeigen()**-Methode ist streng genommen eine sondierende Methode des zweiten Typs:

```
public void anzeigen()
{
   System.out.println("Gewicht = " + gewicht + " kg.");
   System.out.println("Groesse = " + groesse + " cm.");
   System.out.println("Idealgewicht = " + getIdealgewicht() + " kg.");
}
```

Interessant ist hier, dass die Methode **anzeigen()** ihrerseits die Methode **getIdealgewicht()** aufruft. Hier wird dann auch klar, was man unter dem "Zurück geben eines Wertes" versteht. Die sondierende Methode **getIdealgewicht()** wird genau wie ein Attribut aufgerufen. Beim Aufruf wird die Methode ausgeführt, und wenn dann der return-Befehl der Methode kommt, wird **getIdealgewicht()** beendet. In der **anzeigen()**-Methode wird dann für **getIdealgewicht()** der mittels return zurück gegebene Wert eingesetzt.

Schritt 10 - Differenz berechnen

Übung 3.1-3 (3 Punkte)

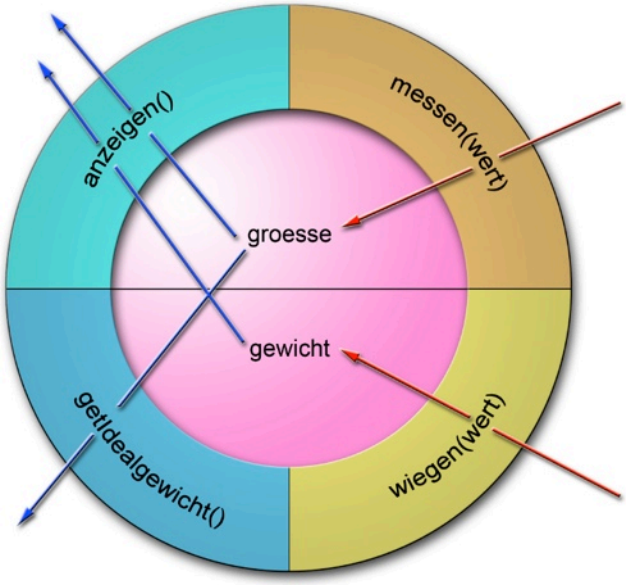
Ergänzen Sie den Quelltext der Klasse **Waage** um die sondierende Methode **getDifferenz()**, welche die Differenz zwischen Idealgewicht und tatsächlichem Gewicht zurück liefert.

Ergänzen Sie dann die Methode **anzeigen()** so, dass nach dem Idealgewicht auch die Differenz zwischen Gewicht und Idealgewicht ausgegeben wird.

Ergänzungsübung (2 Punkte)

Wenn diese Differenz negativ ist, sieht die Ausgabe der **anzeigen()**-Methode nicht so schön aus. Überlegen Sie sich einen Weg, wie man das negative Vorzeichen in der Ausgabe verschwinden lassen kann.

Theorie - Datenkapselung



3.1 - 14 *Eine Datenkapsel*

Wenn die beiden Attribute **groesse** und **gewicht** mit dem Schlüsselwort `private` versehen worden sind...

```
private double groesse, gewicht;
```

... dann kann man von außerhalb der Klasse nicht mehr auf diese Attribute zugreifen, sie sind nach außen hin versteckt. Man spricht auch von "**information hiding**". Man sagt auch, die Attribute stecken im Innern einer **Datenkapsel**. Solche versteckten oder gekapselten Attribute können nur mithilfe **sondierender Methoden** gelesen werden (**get-Methoden**). Hier ein Beispiel:

```
public double getGroesse()
{
   return groesse;
}
```

Solche get-Methoden dürfen dann natürlich *nicht* mit dem Wort `private` versehen werden. Dann wären sie nämlich nicht von außen sichtbar und könnten nicht aufgerufen werden.

Wenn man den Wert eines Attributes verändern will, so geht das nur über eine **manipulierende Methode**. Dabei spielen **Parameter** eine wichtige Rolle; sie informieren die manipulierende Methode nämlich darüber, wie der Attributwert verändert werden soll:

```
public void messen(int gr)
{
   groesse = gr;
}
```

Manipulierende Methoden, die einen Attributwert durch den Wert des Parameters ersetzen, heißen übrigens **set-Methoden** - in Analogie zu den get-Methoden. Obige Methode hätte man also auch **setGroesse(int gr)** nennen können.

Workshop

3.2 Eingabe über die Konsole (fakultativ)

Betrachten Sie den folgenden Quelltext. Was fällt Ihnen auf?

```
import java.util.*;

public class Waage
{
   Scanner eingabe = new Scanner(System.in);
   double gewicht;
   double groesse;

   public Waage()
   {
      gewicht = 0;
      groesse = 0;
   }

   public void wiegen()
   {
     System.out.println("Geben Sie ihr Gewicht in kg ein (xxx,x) : ");
     gewicht = eingabe.nextDouble();
   }

   public void messen()
   {
     System.out.println("Geben Sie ihre Koerpergroesse in cm ein (xxx) : ");
     groesse = eingabe.nextDouble();
   }

   public double idealgewichtBerechnen()
   {
     return (groesse-100)*0.9;
   }

   public double differenzBerechnen()
   {
     return gewicht - idealgewichtBerechnen();
   }

   public void anzeigen()
   {
     System.out.println("Gewicht      : " + gewicht + " kg.");
     System.out.println("Körpergröße  : " + groesse + " cm.");
     System.out.println("Idealgewicht : " + idealgewichtBerechnen() + " kg.");
     System.out.println("Differenz    : " + differenzBerechnen() + " kg.");
   }

}
```

Wenn Sie die Methode **wiegen()** aufrufen, erscheint in der Konsole die Meldung:

```
Geben Sie ihr Gewicht in kg ein (xxx,x) :
```

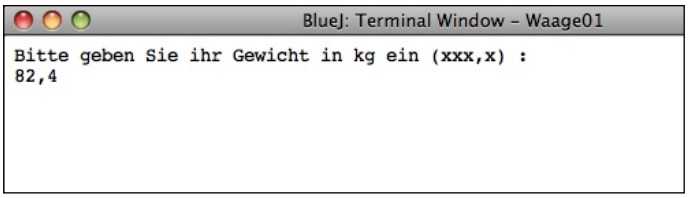
Hinter dem Doppelpunkt blinkt wartend der Textcursor. Sie können jetzt - nein - Sie müssen jetzt eine Zahl eintippen, und zwar eine reelle Zahl in dem Format xxx,x. Wenn Sie 78,30 eintippen, ist das gut. Wenn Sie jetzt denken, dass eine englische Programmiersprache wie Java ein englisches oder amerikanisches Zahlenformat verlangt, dann liegen Sie falsch. Sie dürfen nicht 78.30 eintippen, ein Dezimalpunkt wird von der Methode **nextDouble()** des Objektes **eingabe** als Fehler angesehen. Und wenn Sie irgendetwas anderes als eine Zahl eintippen, ist das sowieso falsch. Ihr Programm bricht dann mit einer Fehlermeldung ab.

Damit diese bequeme aber fehleranfällige Art der Konsoleneingabe funktioniert, müssen Sie zunächst ein Objekt der Klasse **Scanner** deklarieren. Dies geschieht dort, wo auch die anderen Attribute der Klasse **Waage** deklariert werden:

```
    Scanner eingabe = new Scanner(System.in);
```

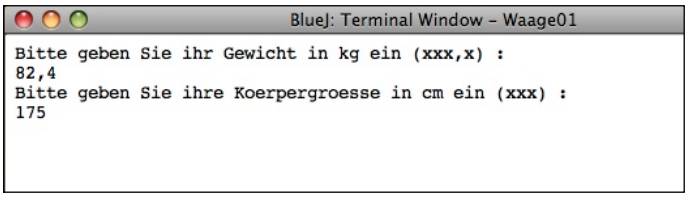
Der Konstruktor der Klasse **Scanner** verlangt als Parameter einen Hinweis auf die Art und Weise, mit der Sie Daten einzugeben gedenken. `System.in` ist die "normale" Eingabe von Zeichen über die Tastatur. Andere Eingabequellen wären beispielsweise Textdateien oder andere Dateien, was uns hier aber nicht weiter interessieren muss.

Die Bedienung dieses Programms erfolgt nun so, dass Sie zunächst ein Objekt der Klasse **Waage** anlegen. Dann rufen Sie mit der rechten Maustaste die Methode **wiegen()** auf. Die Konsole sieht dann so aus:



***3.2 - 1** Eingabekonsole für das Wiegen*

Anschließend rufen Sie - wieder mit der rechten Maustaste - die Methode **messen()** auf. Nach der Eingabe einer Körpergröße sieht die Konsole so aus:



***3.2 - 2** Nach dem Messen*

Nun rufen Sie die Methode **anzeigen()** auf, und Sie erhalten die gewünschte Ausgabe. Wenn Sie sich allerdings auch nur einmal vertippen bei der Eingabe der beiden Werte, so stürzt Ihr Programm gnadenlos ab, und wenn Sie Pech haben, müssen Sie nicht nur BlueJ neu starten, sondern auch ihren Rechner.

Damit sollten Ihnen die Vor- und Nachteile dieser Konsolen-Eingabe klar sein. Man kann sie benutzen, aber die Eingabe von Werten mittels Parameter und BlueJ-Dialogbox ist auch sehr schön.

Konsoleneingabe

In "richtigen" Java-Programmen muss die Eingabe von Daten auch dann möglich sein, wenn die Entwicklungsumgebung BlueJ nicht zur Verfügung steht. Eine Möglichkeit zur Eingabe über die Konsole wurde im letzten Unterrichtsvorhaben aufgezeigt. Hier noch einmal eine kurze Anleitung, was zu tun ist.

Als erstes muss ein Attribut der Klasse **Scanner** erzeugt werden:

```
Scanner eingabe = new Scanner(System.in);
```

Dann kann es eigentlich schon losgehen. Eine double-Zahl kann man mit der Methode **nextDouble()** einlesen:

```
gewicht = eingabe.nextDouble();
```

Für int-Zahlen und andere Datentypen stehen entsprechende Methoden zur Verfügung, beispielsweise **nextInt()** und **nextBoolean()**.

Konsoleneingabe für Experten

Bei der Konsoleneingabe wird ein **Laufzeitfehler** produziert, wenn man zum Beispiel **nextDouble()** abfragt, in dem eingelesenen String aber keine Ziffern vorhanden sind, die als Zahl interpretiert werden könnten.

In diesem Falle kann man aber mit Methoden wie **hasNextDouble()** vorher überprüfen, ob der String überhaupt eine double-Zahl enthält:

```
if (eingabe.hasNextDouble())
   gewicht = eingabe.nextDouble();
else
   gewicht = 0;
```

Workshop und Theorie

3.3 Fallunterscheidungen (I2,I3 / A)

Schritt 1 - Ein gewaltiger Schritt nach vorn

Allzu überzeugend ist die Waage noch nicht. Sie kann das Gewicht feststellen, die Größe messen, das Idealgewicht berechnen und die Differenz zwischen Idealgewicht und gemessenem Gewicht ermitteln, und sie kann diese vier Daten in einem Terminalfenster anzeigen. Wie schön wäre es doch, wenn die Waage auch anzeigen könnte: `"Sie sind zu schwer!"` oder `"Sie sind zu leicht!"` oder sogar: `"Sie müssen 24.3 kg abnehmen!"` bzw. `"Sie müssen 6.4 kg zunehmen!"`.

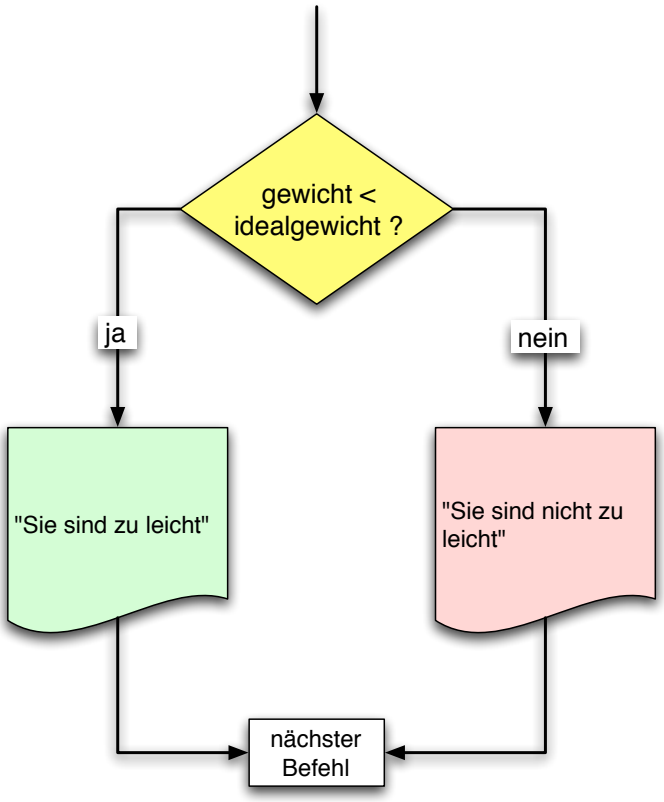
Beginnen wir mit der einfacheren Aufgabe. Die Waage soll ausgeben, ob der Benutzer zu schwer oder zu leicht ist. Verändern Sie dazu die Methode **anzeigen()** wie folgt:

```
public void anzeigen()
{
   System.out.println("Gewicht      :" + gewicht);
   System.out.println("Körpergröße  :" + groesse);

   if (gewicht < idealgewichtBerechnen())
      System.out.println("Sie sind zu leicht");
   else
      System.out.println("Sie sind nicht zu leicht");
}
```

Angenommen, Ihr Idealgewicht beträgt 75 kg, und Ihr tatsächliches Gewicht 72 kg. Dann gibt das Programm aus: `"Sie sind zu leicht"`. Beträgt Ihr tatsächliches Gewicht jedoch 80 kg, so gibt das Programm aus: `"Sie sind nicht zu leicht"`. Das ist noch nicht besonders elegant; es wäre schön, wenn der Computer in diesem Fall `"Sie sind zu schwer"` ausgeben würde. Mit der oben formulierten **if-else-Anweisung** kann man aber nur eine **zweiseitige Fallunterscheidung** bzw. eine **zweiseitige Auswahl** treffen: Entweder ist das Gewicht kleiner als das Idealgewicht, oder es ist nicht kleiner (also gleich oder größer).

Schritt 2 - zweiseitige Auswahl

Stellen wir die in der **anzeigen()**-Methode verwendete Fallunterscheidung einmal graphisch dar:



3.3 -1 Eine zweiseitige Fallunterscheidung

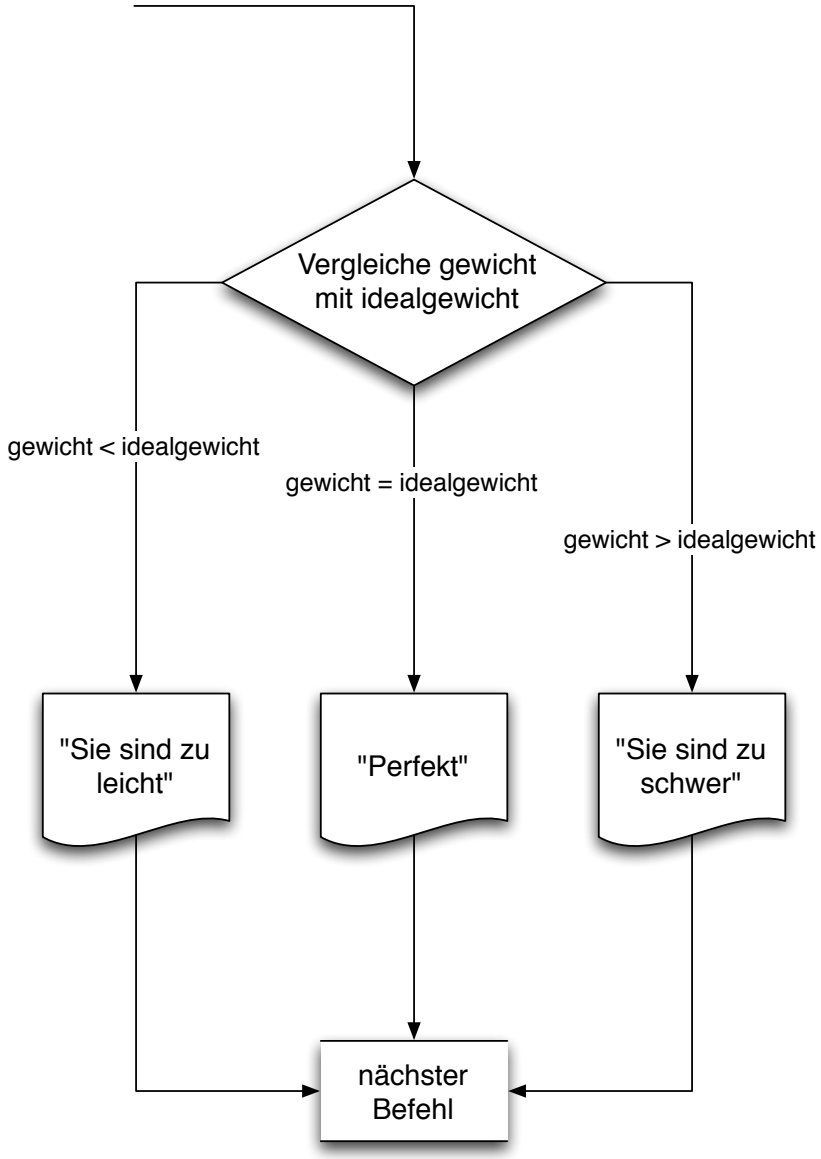
Die Abbildung zeigt ein **Flussdiagramm**; eine graphische Darstellung des Programmflusses. Bisher verliefen unsere Programme streng linear, also ohne Abzweigungen. Jetzt sehen wir zum ersten Mal ein verzweigtes Programm. Der **Programmfluss** teilt sich, jenachdem, ob die definierte **Bedingung** zutrifft. Ist die Bedingung **gewicht < idealgewicht** gültig (wahr, true), so verzweigt der Programmfluss nach links. Ist die Bedingung dagegen nicht gültig (falsch, false), so verzweigt der Programmfluss nach rechts.

In den meisten Programmiersprachen implementiert man eine solche zweiseitige Auswahl mithilfe einer if-else-Anweisung:

```
if (gewicht < idealgewichtBerechnen())
   System.out.println("Sie sind zu leicht");
else
   System.out.println("Sie sind nicht zu leicht");
```

Schritt 3 - dreiseitige Auswahl

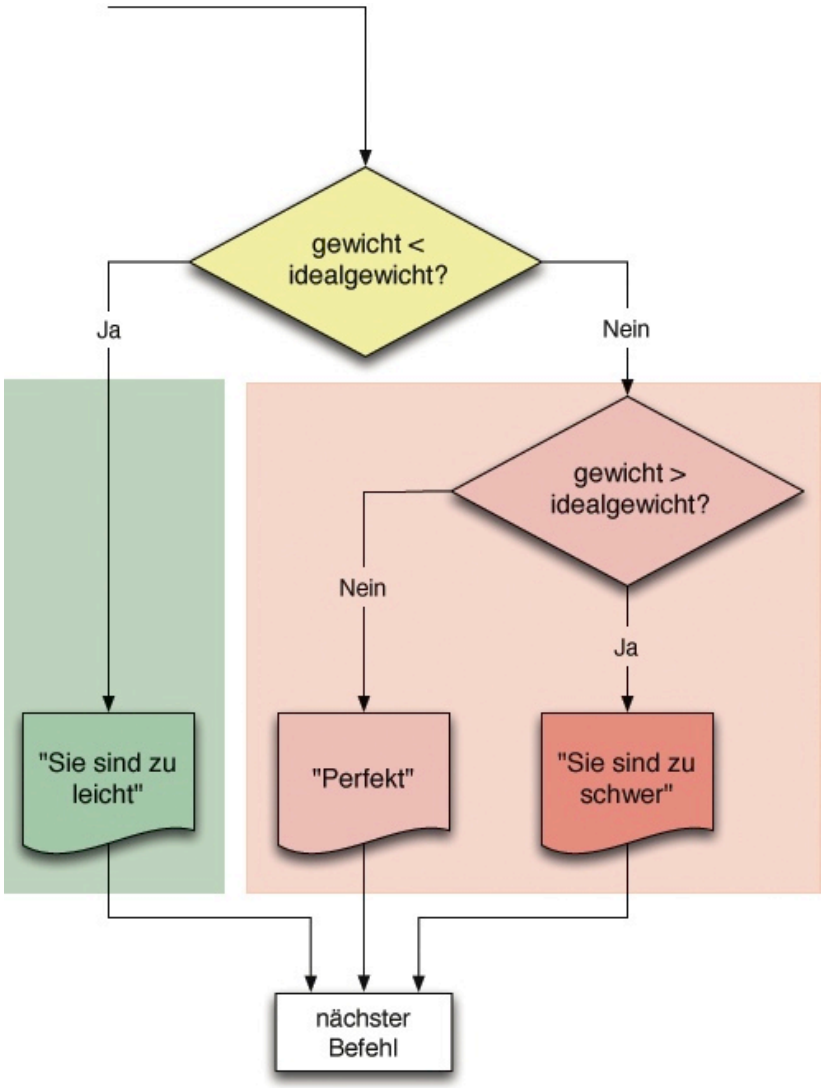
So richtig gut ist die Ausgabe und die Beurteilung des Gewichts immer noch nicht. Was passiert zum Beispiel, wenn die Person weder zu leicht noch zu schwer ist, sondern das Idealgewicht hat? Wir benötigen eine **dreiseitige Auswahl**. Graphisch könnte man diese so darstellen:



***3.3 - 2** Eine dreiseitige Fallunterscheidung - leider* so *nicht möglich*

Leider gibt es keine Programmiersprache, in der ein dreiseitige Auswahl so realisiert werden kann wie in der Abbildung. In einem Prozessor kann nämlich nur der Inhalt von *zwei* Registern verglichen werden, nicht aber der Inhalt von *drei* Registern. Daher kann auch der if-else-Befehl nur zweiseitige Vergleiche treffen.

Wenn wir eine dreiseitige Auswahl implementieren wollen, müssen wir mehrere if-Befehle verwenden. Wie das geht, macht Ihnen die folgende Abbildung klar.



3.3 - 3 *Eine dreiseitige Auswahl, aufgebaut aus zwei zweiseitigen Auswahlen*

Und so realisiert man eine **dreiseitige Auswahl**:

```
if (gewicht < idealgewichtBerechnen())
    System.out.println("Sie sind zu leicht");
else
    if (gewicht > idealgewichtBerechnen())
       System.out.println("Sie sind zu schwer");
    else
       System.out.println("Perfekt");
```

3.3 - 4 Quelltext einer dreiseitigen Auswahl

Die **äußere if-else-Anweisung** überprüft die Bedingung

```
(gewicht < idealgewichtBerechnen() )
```

Ist diese Bedingung erfüllt, so wird der **if-Zweig** ausgeführt, der durch den oberen (grünen) Kasten repräsentiert wird. Hier besteht der if-Zweig aus einer **System.out.println()**-Anweisung. Ist die Bedingung jedoch nicht erfüllt, wird der else-Zweig ausgeführt.

Der **else-Zweig**, der durch den unteren (roten) Kasten repräsentiert wird, besteht nicht aus einer simplen Anweisung, sondern aus einer weiteren if-else-Anweisung. Diese **innere if-else-Anweisung** besteht genau wie die äußere aus einer Bedingung, einem if-Zweig und einem else-Zweig. Beide Zweige sind hier nicht sehr kompliziert, sondern bestehen jeweils aus einer einfachen Anweisung.

Spielen wir das Ganze noch einmal konkret durch. Das Idealgewicht der Person sei 75 kg.

Fall 1: Gewicht = 70 kg

Die Bedingung der äußeren if-else-Anweisung ist erfüllt, daher wird der if-Zweig (oberer Kasten) ausgeführt. Es wird ausgegeben: "Sie sind zu leicht".

Fall 2: Gewicht = 80 kg

Die Bedingung der äußeren if-else-Anweisung ist *nicht* erfüllt, daher wird der else-Zweig (unterer Kasten) ausgeführt. Dieser besteht aus einer weiteren if-else-Bedingung. Da diese erfüllt ist, wird der if-Zweig der inneren if-else-Anweisung ausgeführt: "Sie sind zu schwer".

Fall 3: Gewicht = 75 kg

Die Bedingung der äußeren if-else-Anweisung ist nicht erfüllt, daher wird der else-Zweig (unterer Kasten) ausgeführt. Dieser besteht aus einer weiteren if-else-Bedingung. Da diese ebenfalls nicht erfüllt ist, wird der else-Zweig der inneren if-else-Anweisung ausgeführt: "Perfekt".

Schritt 4 - Und es geht noch besser

Stellen Sie sich vor, das vom Programm berechnete Idealgewicht sei 75,0000 kg, das reale Gewicht jedoch 75,0001 kg. Eine normale Waage würde diesen Unterschied von 100 mg überhaupt nicht bemerken. Unser Java-Programm würde in diesem Fall jedoch behaupten, dass Sie zu schwer sind. Die Bedingungen `(gewicht < idealgewicht())` bzw. `(gewicht > idealgewicht())` nimmt das Programm *wörtlich*. Wir wollen diese Fallunterscheidung nun etwas *toleranter* machen, so dass erst ab einer Gewichtsdifferenz von 1 kg eine entsprechende Meldung kommt.

Schauen Sie sich die entsprechende Verbesserung der Methode an:

```
public void ausgeben()
{
   System.out.println("Gewicht      :" + gewicht);
   System.out.println("Körpergröße  :" + groesse);

   if (differenzBerechnen() <= -1)
      System.out.println("Sie sind zu leicht!");

   else if (differenzBerechnen() >= 1)
      System.out.println("Sie sind zu schwer!");

   else
      System.out.println("Sie sind perfekt");
}
```

Wenn die Person mindestens 1 kg unter dem Idealgewicht liegt:

```
if (differenzBerechnen() <= -1)
```

so wird ausgegeben `"Sie sind zu leicht!"`. Trifft dieser Fall nicht zu, so tritt der "else-Fall" der äußeren if-else-Anweisung ein, der aus der inneren if-else-Anweisung besteht:

```
if (differenzBerechnen() >= 1)
```

Wenn dieser Fall vorliegt, so wird ausgegeben `"Sie sind zu schwer!"`. Wenn auch dieser Fall nicht vorliegt, so tritt der else-Fall der inneren if-else-Anweisung ein.

Theorie

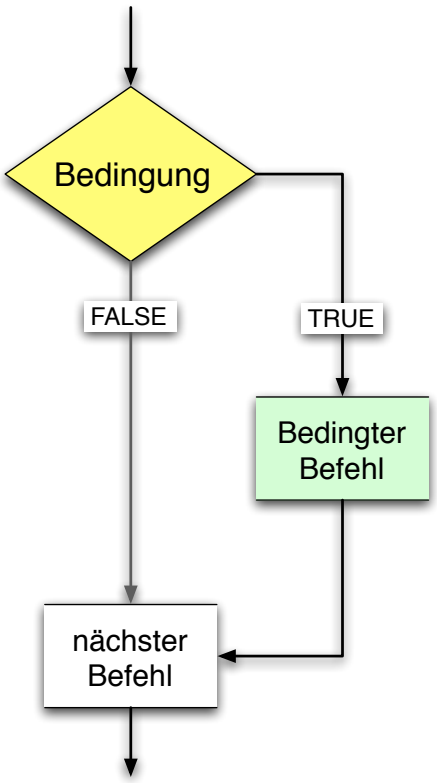
3.4 Die if-Anweisung (I2,I3 / A)

Allgemeines

Betrachten Sie die folgende Anweisung etwas näher:

```
if (zaehler != 0)
    ergebnis = nenner / zaehler;
```

Wenn der Zähler ungleich Null ist, **dann** wird die Division durchgeführt. Ansonsten passiert gar nichts. Daher könnte man diese Anweisung auch als **Wenn-Dann-Anweisung** bezeichnen. Das allgemeine Flussdiagramm für eine solche Wenn-Dann-Anweisung sieht so aus:



3.4 - 1 Flussdiagramm für eine if-Anweisung

Wenn die Bedingung erfüllt ist (TRUE), dann und nur dann wird der **bedingte Befehl** ausgeführt. Anschließend wird der nächste Befehl ausgeführt, der nach der Wenn-Dann-Anweisung kommt.

Wenn die Bedingung jedoch nicht erfüllt ist (FALSE), wird sofort mit dem Befehl weitergemacht, der nach der if-Anweisung kommt.

Die if-Anweisung in Java

Was wir hier haben, nennt man in der Programmierung eine **einseitige Auswahl**. Einseitige Auswahlen können in jeder wichtigen Programmiersprache realisiert werden. Hier ein paar Beispiele:

Visual Basic:

```
IF Bedingung THEN
   Anweisungsliste
END IF
```

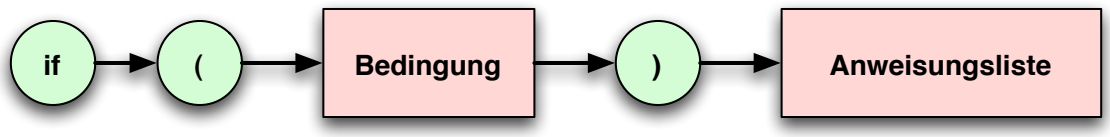
Pascal:

```
IF Bedingung THEN Anweisungsliste;
```

Java:

```
if ( Bedingung ) Anweisungsliste;
```

Die nächste Abbildung zeigt den allgemeinen Aufbau einer Java-if-Anweisung in Form eines **Syntaxdiagramms**:



3.4 - 2 Syntaxdiagramm der if-Anweisung

Zunächst kommt das Schlüsselwort `if`, dann kommt, in runden Klammern `( )` stehend, eine **logische Bedingung**. Schließlich kommt eine so genannte **Anweisungsliste**.

if-Anweisung

Allgemeine Syntax in der Sprache Java:

```
if (bedingung) anweisungsliste
```

Nur wenn die Bedingung erfüllt ist, wird der Befehl ausgeführt. Wenn die Bedingung nicht erfüllt ist, so wird der Befehl übersprungen.

Die if-Anweisung wird auch als **einseitige Auswahl** oder **einseitige Verzweigung** oder **einseitige Fallunterscheidung** bezeichnet.

Einfache Bedingungen

Eine **Bedingung** ist ein Java-Ausdruck, der entweder wahr (`true`) oder falsch (`false`) ist. Schauen wir uns mal ein paar solcher Bedingungen an:

```
x > y
```

Hier handelt es sich um einen einfachen **Vergleich**. Wenn der Wert der Variable `x` größer ist als der Wert der Variable `y`, dann ist diese logische Bedingung wahr. Andernfalls (`x = y` oder `x < y`) ist die Bedingung nicht erfüllt oder falsch.

```
0 <= eingabe
```

Die Bedingung ist wahr, wenn der Wert der Variable `eingabe` Null oder größer als Null ist. Hat `eingabe` aber beispielsweise den Wert -4, dann ist die Bedingung nicht erfüllt.

Allerdings sollte man hier kritisch anmerken, dass die Formulierung der Bedingung nicht sehr anschaulich ist, sondern den Betrachter eher verwirrt. Besser wäre die folgende Formulierung:

```
eingabe > 0
```

Auch auf solche "psychologischen" Aspekte sollte man bei der Formulierung von Bedingungen achten, um sich selbst und anderen die Arbeit zu vereinfachen.

Zusammengesetzte Bedingungen

Eine Routineaufgabe bei der Programmierung ist die Überprüfung von Variablenwerten. Es soll beispielsweise festgestellt werden, ob die Variable `eingabe` in dem Intervall `[0, 10]` liegt. Dazu muss man zwei Vergleiche durchführen. Zunächst muss überprüft werden, ob `eingabe` größer oder gleich Null ist:

```
0 <= eingabe
```

Zweitens muss überprüft werden, ob `eingabe` kleiner oder gleich 10 ist:

```
eingabe <= 10
```

Erst wenn *beide* Bedingungen erfüllt sind, ist auch die zusammengesetzte Bedingung *"eingabe liegt im Intervall [0, 10]"* erfüllt. In Java heißt die entsprechende Formulierung dieser Bedingung:

```
(0 <= eingabe) && (eingabe <= 10)
```

Der UND-Operator wird durch die beiden kaufmännischen &-Zeichen realisiert. Wichtig ist die Klammerung der beiden einfachen Bedingungen. Würde man schreiben:

```
0 <= eingabe && eingabe <= 10
```

so würde das Programm diese Angabe folgendermaßen interpretieren:

```
0 <= (eingabe && eingabe) <= 10
```

Der &&-Operator hat nämlich eine höhere Priorität als die Vergleichsoperatoren <= und >=. Ähnlich ist es ja, wen wir eine Rechenoperation wie

```
4 + 5 * 6 + 7
```

aufschreiben würden. Es würden nicht 9 und 13 multipliziert, sondern 4, 30 und 7 würden addiert, da zunächst die Multiplikation ausgeführt würde, weil der *-Operator eine höhere Priorität hat als der +-Operator (*"Punktrechnung geht vor Strichrechnung"*). Bei arithmetischen Ausdrücken löst man dieses Problem mithilfe von Klammern:

```
(4 + 5) * (6 + 7)
```

Und entsprechend muss man Klammern bei zusammengesetzten Bedingungen benutzen.

Betrachten wir nun eine recht komplizierte zusammengesetzte Bedingung:

```
(((0 <= eingabe) && (eingabe <= 10)) || (eingabe >= 100))
```

Diese Bedingung ist erfüllt, wenn die Eingabe entweder im Intervall [0, 10] liegt oder wenn sie mindestens den Wert 100 hat. Der logische **oder-Operator** besteht aus zwei senkrechten Strichen `||`. Achten Sie hier wieder auf die Klammerung.

Vergleichsoperatoren

In Java gibt es die folgenden Vergleichsoperatoren:

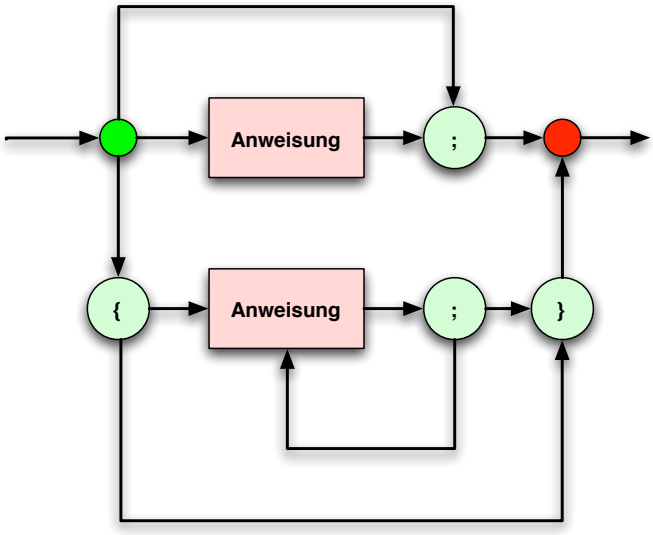
`==`	gleich
`!=`	ungleich
`<`	kleiner
`<=`	kleiner oder gleich
`>`	größer
`>=`	größer oder gleich

Logische Operatoren

Einfache Vergleiche können mit folgenden Operatoren zu zusammengesetzten Vergleichen verknüpft werden:

`&&`	logisches UND bzw. AND
`||`	logisches ODER bzw. OR (kein exklusives ODER bzw. XOR)
`!`	logisches NICHT bzw. NOT

Anweisungslisten



3.4 - 3 Syntaxdiagramm für eine Anweisungsliste

Das obige Syntaxdiagramm ist folgendermaßen zu lesen.

Eine Anweisungsliste kann leer sein (direkter Weg vom grünen Startpunkt zum Semikolon).
Ein Beispiel für eine if-Anweisung mit einer leeren Anweisungsliste:

```
if (x < y);
```

Laut Syntaxdiagramm ist eine solche if-Anweisung erlaubt. Ob sie Sinn macht, ist eine andere Frage.

Eine Anweisungsliste kann aus einer einzigen Anweisung (und einem Semikolon) **bestehen.**
Hier ein Beispiel:

```
if (x < 0) x = 0;
```

Wenn x einen negativen Wert hat, wird x wieder auf Null gesetzt. Eine sehr nützliche Routine, wenn zum Beispiel ein Graphik-Objekt in einer Animation den linken Bildschirmrand verlassen will.

Eine Anweisungsliste kann aus mehreren Anweisungen bestehen, die in Klammern eingeschlossen sind.
Hinter jeder Anweisung muss ein Semikolon stehen. Hier ein Beispiel:

```
if (x < 0)
{
   x = 0;
   y = 0;
}
```

Hinter der schließenden Klammer steht kein Semikolon!

Eine Anweisungsliste kann aus einem leeren Klammerpaar bestehen:

```
if (x < 0) { }
```

Hinter der schließenden Klammer steht auch hier kein Semikolon!

Theorie

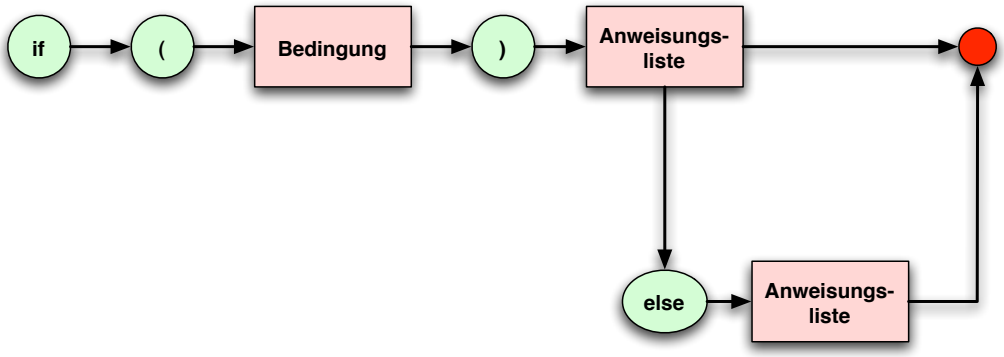
3.5 Die if-else-Anweisung (I2,I3 / A)

In den meisten Programmiersprachen gibt es neben der einseitigen Auswahl auch die zweiseitige Auswahl. Schauen wir uns zunächst ein Beispiel an:

```
if (x >= 20)
   x = x - 5;
else
   x = x + 5;
```

Wenn `x` mindestens den Wert 20 hat, so wird `x` um 5 dekrementiert (erniedrigt). Ist diese Bedingung nicht erfüllt, hat `x` also zum Beispiel den Wert 17, so wird `x` um 5 inkrementiert (erhöht). Dieser "andere Fall" wird durch das Schlüsselwort `else` eingeleitet.

Betrachten wir das Syntaxdiagramm der if-else-Anweisung:



***3.5 - 1** Syntaxdiagramm der if-else-Anweisung*

Die if-else-Anweisung besteht zunächst aus einer einfachen if-Anweisung (oben im Bild). Wenn die Anweisungsliste beendet ist, *muss* nichts mehr kommen; es *kann* aber auch das Schlüsselwort else folgen, und dann *muss* eine zweite Anweisungsliste kommen.

Hier ein paar Beispiele für if-else-Anweisungen:

```
if (x > y)
   System.out.println(x + " ist größer als "+y);
else
  System.out.println(x + " ist nicht größer als "+y);
```

In diesem ersten Beispiel bestehen beide Anweisungslisten aus je einem einfachen Befehl, der von einem Semikolon abgeschlossen wird.

```
if (gewicht < idealgewichtBerechnen())
{
   System.out.println("Wie haben Sie das denn gemacht?");
   System.out.println("Sie sind viel zu leicht!");
   System.out.println("Bitte nehmen Sie "+ differenzBerechnen() + " kg zu!");
}
else
   System.out.println("Sie müssen " + differenzBerechnen() + " kg abnehmen!");
```

Dies ist ein schon etwas komplexeres Beispiel. Die Anweisungsliste des if-Zweiges besteht aus drei Befehlen, die dann natürlich geklammert werden müssen. Die Anweisungsliste des else-Zweiges besteht dagegen aus nur einem Befehl, Klammern sind daher nicht nötig. Laut Syntaxdiagramm für die Anweisungsliste sind Klammern jedoch syntaktisch nicht falsch, auch wenn nur ein Befehl kommt.

if-else-Anweisung

Allgemeine Syntax in der Sprache Java:

```
if (bedingung)
   befehl1;
else
   befehl2;
```

Wenn die logische Bedingung erfüllt ist wird der Befehl 1 ausgeführt. Andernfalls (else) wird der Befehl 2 ausgeführt. Einer der beiden Befehle wird aber auf jeden Fall ausgeführt.

Die if-else-Anweisung wird auch als **zweiseitige Auswahl**, **zweiseitige Verzweigung** oder **zweiseitige Fallunterscheidung** bezeichnet.

Für Abiturienten

3.6 Allgemeines zu Syntaxdiagrammen (I3 / A,D)

Syntaxdiagramme veranschaulichen, nach welchen Regeln man bestimmte Ausdrücke in einer Programmiersprache erstellen muss. Um beispielsweise die Syntax einer if-else-Anweisung zu erläutern, kann man

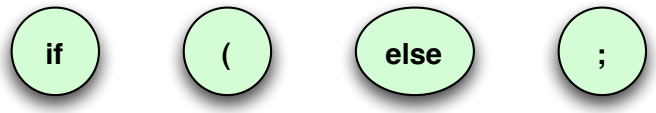
a) die Syntax beschreiben: "Zunächst kommt das Schlüsselwort if, dann folgt eine logische Bedingung, die in runden Klammern stehen muss. Danach kommt dann eine Anweisungsliste. Jetzt *kann* die if-else-Bedingung zu Ende sein; es kann aber auch das Schlüsselwort else gefolgt von einer weiteren Anweisungsliste folgen."

b) eine Reihe von Beispielen geben, wie wir das bereits häufiger gemacht haben,

c) ein Syntaxdiagramm zeichnen, auch das kennen Sie bereits.

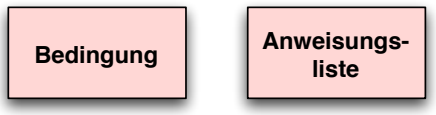
Syntaxdiagramme bestehen immer aus drei Komponenten, die unterschiedlich (aber in der Literatur leider nicht einheitlich) dargestellt werden.

Terminalsymbole oder **Terminale**



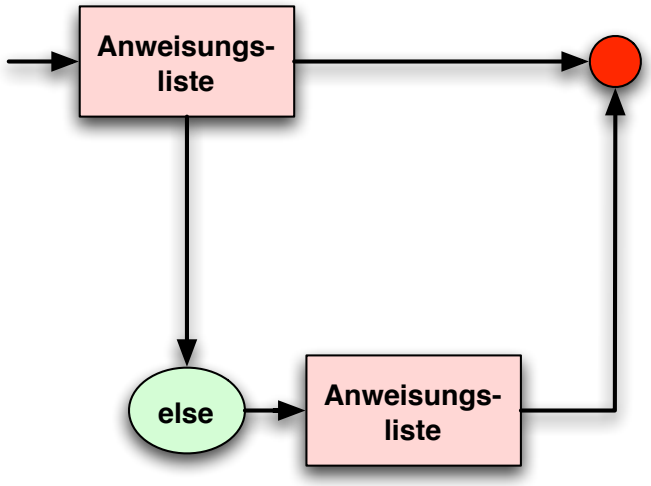
Terminale sind Zeichen oder Zeichenketten, die *genau* so geschrieben werden müssen, wie sie im Syntaxdiagramm stehen. Die Schlüsselworte `if` und `else` gehören z.B. zu diesen Terminalen, aber auch Zeichen wie Klammern, Semikolons, Plus- und Minuszeichen etc.

Nichtterminalsymbole oder **Nichtterminale**



Nichtterminale sind Konstrukte, die durch ein Syntaxdiagramm definiert werden müssen. Das Syntaxdiagramm für das Nichtterminal **Anweisungsliste** haben Sie zum Beispiel in der Abbildung 3.5 - 1 kennen gelernt.

Pfeile

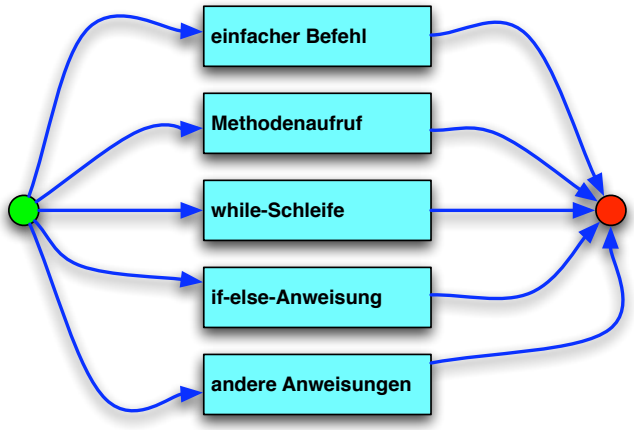


Die **Pfeile** verbinden die Terminale und Nichtterminale des Syntaxdiagramms miteinander. Dabei können auch Verzweigungen auftreten. Oben sehen Sie einen Ausschnitt aus dem Syntaxdiagramm für die if-else-Anweisung. Nach der ersten Anweisungsliste ist entweder Schluss (durch den roten Punkt symbolisiert), oder es kommt das Terminal `else`, gefolgt von einer Anweisungsliste.

Theorie

3.7 Geschachtelte if-else-Anweisungen (I2,I3 / A)

Das einzige Nichtterminal, das in dem Syntaxdiagramm für die Anweisungsliste auftaucht, ist "Anweisung". Unter einer **Anweisung** verstehen wir einen einfachen Befehl oder Methodenaufruf wie z.B. **System.out.println()**, eine while-Anweisung (wird später noch behandelt), eine for-Anweisung (wird auch später noch behandelt) oder eben eine if-else-Anweisung.



3.7 - 1 Syntaxdiagramm für eine Anweisung

Wie sieht das aus, wenn die Anweisungsliste einer if-Anweisung aus einer if-Anweisung besteht? Konstruieren wir doch mal ein Beispiel:

```
if (x < 20)
   if (y < 20)
      z = x + y;
```

Wir haben hier eine geschachtelte if-Anweisung erzeugt. Kenner der Materie hätten hier natürlich einfach geschrieben:

```
if ((x < 20) && (y < 20))
   z = x + y;
```

Beide Anweisungen sind im Prinzip gleichwertig. Zunächst wird überprüft, ob `x < 20` ist. Wenn diese Bedingung *nicht* erfüllt ist, wird der Test `y < 20` gar nicht erst durchgeführt.

Konstruieren wir nun eine if-Anweisung, die eine if-else-Anweisung enthält:

```
if (x < 20)
   if (y < 20)
      z = x + y;
   else
      z = x - y;
```

Das else bezieht sich auf die innere if-Anweisung. Wenn also y nicht kleiner als 20 ist, dann wird der Befehl hinter dem else ausgeführt. Wenn jedoch x nicht kleiner als 20 ist, so passiert gar nichts, denn für die äußere if-Anweisung existiert ja kein else-Zweig. Wir haben es also mit einer **dreiseitigen Auswahl** zu tun, wie wir sie bereits im letzten Unterrichtsvorhaben behandelt haben.

Dreiseitige Auswahl

Durch Verschachtelung zweier if-else-Anweisungen kann man eine dreiseitige Auswahl konstruieren. Allgemeine Syntax in Java:

```
if (bedingung1)
   befehl1;
else if (bedingung2)
   befehl2;
else
   befehl3;
```

Für eine **fünfseitige Auswahl** stellen wir uns ein Ratespiel vor. Der Computer hat eine Zufallszahl `z` ermittelt, und der Benutzer hat eine Ratezahl `r` eingegeben. Das Programm soll nun eine differenzierte Ausgabe machen, wobei fünf Fälle unterschieden werden sollen:

```
if (r < z/2)
   ausgabe("viel zu klein!");
else if (r < z)
   ausgabe("zu klein!");
else if (r == z)
   ausgabe("geraten!");
else if (r > z*2)
   ausgabe("viel zu groß");
else
   ausgabe("zu groß");
```

Bei solchen **mehrseitigen Auswahlen** ist unbedingt auf die korrekte Reihenfolge der einzelnen Abfragen zu achten. Hätte man zum Beispiel am Anfang geschrieben:

```
if (r < z)
   ausgabe("zu klein!");
else if  (r < z/2)
   ausgabe("viel zu klein!");
```

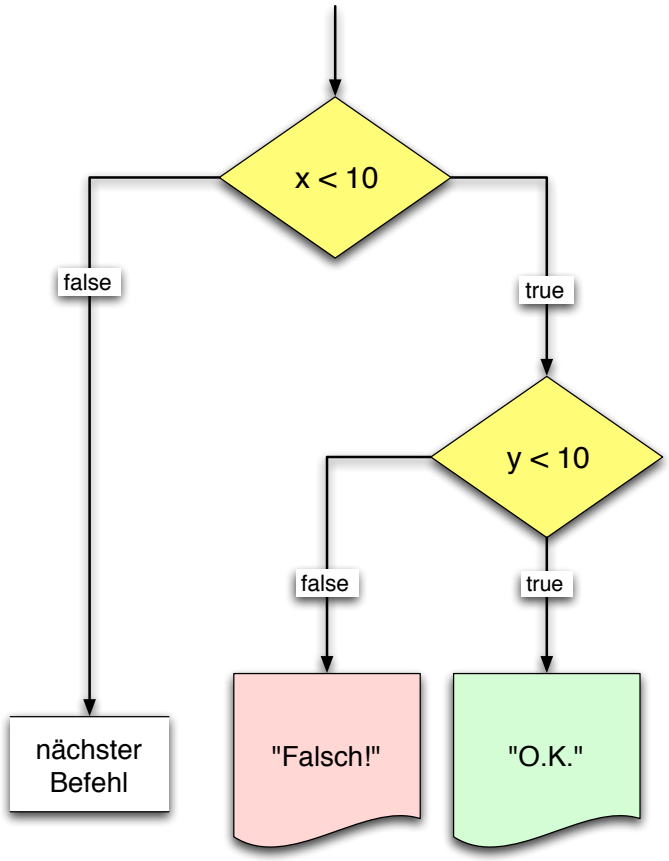
dann würde das Programm bei einer Ratezahl, die die zweite Bedingung `r < z/2` erfüllt, eine falsche Meldung ausgeben. Wenn `r < z/2` ist, dann gilt auch `r < z` (z ist ja doppelt so groß wie z/2). Das heißt, bereits die erste Bedingung `r < z` wäre erfüllt, und somit würde das Programm melden `"zu klein!"`. Es hätte aber `"viel zu klein!"` ausgeben müssen.

Das Dangling-Else-Problem

Betrachten Sie bitte folgenden Quelltext:

```
if (x < 10)
   if (y < 10)
      ausgabe("O.K.")
else
      ausgabe("Falsch!");
```

Was wird hier ausgegeben? Lassen Sie sich nicht von der Stellung des `else` verwirren! Es wird hier der Eindruck erweckt, dass das `else` zum ersten `if` gehört. Das ist aber nicht der Fall, ein `else` gehört immer zum *vorangegangenem* `if`, in unserem Fall also zum zweiten `if`. Die Meldung "falsch" wird also ausgegeben, wenn `y` nicht kleiner als `10` ist.



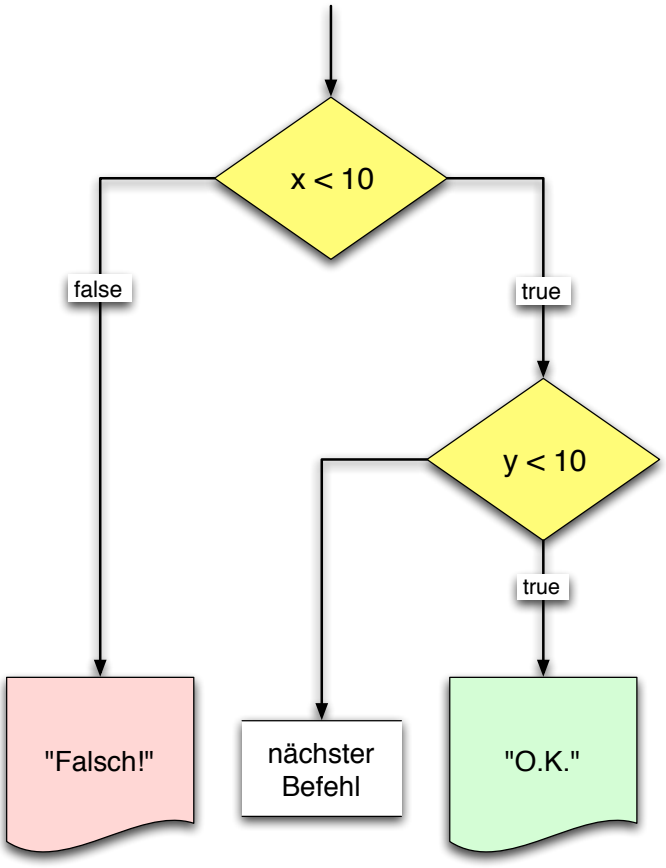
3.7 - 2 Flussdiagramm zum dangling-else-Problem

Dieses hier beschriebene Problem ist als "**dangling-else-problem**" bekannt geworden. Es ist nicht nur von akademischem Interesse, sondern hat auch durchaus praktische Auswirkungen auf die tägliche Programmierarbeit.

Der Quelltext

```
if (x < 10)
{
   if (y < 10)
      ausgabe("O.K.")
}
else
   ausgabe("Falsch!");
```

arbeitet nun völlig anders. Hier wird "Falsch" ausgegeben, wenn `x` nicht kleiner als `10` ist, wenn also die erste if-Bedingung nicht zutrifft. Die geschweiften Klammern haben dafür gesorgt, dass das `else` nicht zum inneren `if` gehört, sondern zum äußeren. Hier haben wir also einen der seltenen Fälle, dass man geschweifte Klammern verwenden muss, obwohl nur ein einziger Befehl ausgeführt wird.



3.7 - 3 Flussdiagramm zum dangling-else-Problem

Anmerkung:

Natürlich hätte man das oben gezeigte Problem auch anders lösen können, nämlich so:

```
if ((x < 10) && (y < 10))
   ausgabe("O.K.);
else
   ausgabe ("Falsch!");
```

3.8 Übungen zum Projekt Waage (I1,I2,I3 / A,M,I,D)

Übung 3.8-1

Verändern Sie Ihre **anzeigen()**-Methode so, dass sie folgendes ausgibt (die Zahlen sind nur Beispiele):

1. Wenn die Differenz zwischen tatsächlichem Gewicht und Idealgewicht 1 kg oder größer ist:
   „Sie haben Übergewicht und wiegen 3.4 kg zu viel!".
2. Wenn die Differenz -1 kg oder kleiner ist:
   „Sie haben Untergewicht und wiegen 5.3 kg zu wenig!"
3. Andernfalls:
   „Sie haben Idealgewicht!"

Übung 3.8-2

Eine weitere Verbesserung des Algorithmus: Bisher wird nur ausgegeben, ob die Person Idealgewicht, Über- oder Untergewicht hat. Verändern Sie den Algorithmus so, dass jetzt *fünf* Fälle unterschieden werden:

1. Starkes Übergewicht (mehr als 10 kg über dem Idealgewicht)
2. Übergewicht (mehr als 1 kg über dem Idealgewicht)
3. Idealgewicht (maximal 1 kg über / unter dem Idealgewicht)
4. Untergewicht (mehr als 1 kg unter dem Idealgewicht)
5. Starkes Untergewicht (mehr als 5 kg unter dem Idealgewicht)

Übung 3.8-3

Mit dem Body-Mass-Index BMI kann man den Ernährungszustand einer Person genauer einschätzen als mit dem Idealgewicht. Die Formel zur BMI-Berechnung ist:

```
BMI = Gewicht / Körpergröße²
```

Das Körpergewicht (in kg) wird durch das Quadrat der Körpergröße (in m) dividiert. Jemand, der 85 kg wiegt und 1,90 m groß ist, hat also einen BMI von 85 / 1,9² = 23,54.

Schreiben Sie eine sondierende Methode, welche nach obigen Vorgaben den BMI berechnet und als double-Zahl zurück liefert.

Übung 3.8-4

Beachten Sie die folgende Tabelle:

Klasse	Kategorie	BMI in kg/m²
1	Starkes Untergewicht	< 16.00
2	Mäßiges Untergewicht	16.00 - 16.99
3	Leichtes Untergewicht	17.00 - 18.49
4	Normalgewicht	18.50 - 24.99
5	Präadipositas	25.00 - 29.99
6	Adipositas Grad I	30.00 - 34.99
7	Adipositas Grad II	35.00 - 39.99
8	Adipositas Grad III	40.00 und mehr

Ein Mann mit dem BMI von 27,7 hätte also bereits eine deutliche Präadipositas (Adipositas = Fettleibigkeit).

Schreiben Sie eine sondierende Methode, welche mithilfe einer ziemlich stark verschachtelten if-else-Anweisung die Gewichtsklasse der Person ermittelt und als int-Wert (1..8) zurück liefert.

3.9 Weitere Übungen zur if-else-Anweisung (I1,I2,I3,I4 / A,M,I,D)

Übung 3.9-1

Schreiben sie ein Programm zur Berechnung einer Steuer. Wenn der zu versteuernde Betrag kleiner als 5000 Euro ist, müssen gar keine Steuern gezahlt werden. Bei einem Betrag ab 5.000 Euro sind 10% Steuern zu zahlen, bei einem Betrag ab 20.000 Euro 15% und bei einem Betrag ab 50.000 Euro sogar 20%.

Eingabe: Der zu versteuernde Betrag

Ausgabe: Der Steuersatz und die zu zahlenden Steuern.

Übung 3.9-2

Schreiben Sie einen Kino-Simulator, der folgendes macht.

Eingabe 1: Altersfreigabe in Jahren für den jeweiligen Film (zum Beispiel 12)

Eingabe 2: Ihr Alter in Jahren (zum Beispiel 17)

Eingabe 3: Vollpreis einer Eintrittskarte (zum Beispiel 6,30 €)

Eingabe 4: Reihe, in der Sie sitzen wollen (zum Beispiel 12)

Ausgabe: Eintrittspreis, falls Sie alt genug für den Film sind, ansonsten Meldung "Sie sind nicht alt genug für den Film".

Der Preis einer Eintrittskarte hängt vom Alter des Besuchers ab: Unter 6 Jahre = halber Vollpreis; 6 - 12 Jahre = ¾ Vollpreis, 13 - 16 Jahre = 10% Ermäßigung, über 16 Jahre = Vollpreis.

Der Preis der Eintrittskarte hängt außerdem von der Reihe ab, in der der Besucher sitzen möchte. Reihen 1 bis 4 (vorne) = 30% Ermäßigung, Reihen 5 bis 8 (mitte) = Preis wie oben berechnet, Reihen 9 bis 11 (hinten) = 30% Zuschlag, Reihe 12 (ganz hinten) = 50% Zuschlag.

Übung 3.9-3 - Eine Waage für Informatiker

Verändern Sie das Programm zur Waage derart, dass das Gewicht im Binärcode angezeigt wird. Statt z.B 77 kg soll angezeigt werden: 01001101.

Falls Sie nicht mehr wissen, was Binärcode ist, recherchieren Sie bitte, was man darunter versteht und wie man Dezimalzahlen in Binärcode umwandelt.

3.10 Die switch-Anweisung (I1,I2,I3,I4 / A,M,I,D)

Fangen wir gleich mit einem etwas komplexeren Beispiel an:

```
public class Alkane
{
    public String gibAlkan(int x)
    {
       String praefix = "";

       switch (x)
       {
          case  1: praefix = "Meth"; break;
          case  2: praefix = "Eth";  break;
          case  3: praefix = "Prop"; break;
          case  4: praefix = "But";  break;
          case  5: praefix = "Pent"; break;
          case  6: praefix = "Hex";  break;
          case  7: praefix = "Hept"; break;
          case  8: praefix = "Oct";  break;
          case  9: praefix = "Non";  break;
          case 10: praefix = "Dec";
       }

       return praefix + "an";
    }
}
```

Die Methode **gibAlkan()** gibt Ihnen die Namen der ersten zehn Alkane zurück. Die Zahl der Kohlenstoff-Atome wird als Parameter `x` übergeben, und dann entscheidet die `switch`-Anweisung darüber, welches die korrekte Vorsilbe für das Alkan ist. Am Ende wird dann die Silbe "an" angehängt.

Die `break`-Anweisungen sind notwendig, damit zum Ende der `switch`-Anweisung gesprungen werden kann, wenn die jeweilige Bedingung zutrifft. Ohne die `break`-Anweisungen würde immer die Vorsilbe "Dec" gewählt werden, egal, wie groß `x` ist.

Die switch-Anweisung funktioniert allerdings nur mit ordinalen Datentypen wie byte, int oder char, die im Rechner durch ganze Zahlen dargestellt werden.

Wäre die Variable `x` vom Typ `float` oder `double` müsste man eine zehnfach geschachtelte if-else-Anweisung verwenden.

Auch die `switch`-Anweisung kennt eine Art else-Zweig, nur heißt hier das Schlüsselwort nicht "else", sondern "default":

```
switch(n)
{
   case 1 : System.out.println("Nummer Eins!"); break;
   case 2 : System.out.println("Nummer Zwei!"); break;
   default: System.out.println("Bitte geben Sie 1 oder 2 ein!"); break;
}
```

Übung 3.10-1 (einfache Variante)

Schreiben Sie eine Klasse **OrganischeVerbindung**, die ähnlich arbeitet wie die Klasse **Alkane** im letzten Abschnitt. Allerdings soll hier nicht nur die Zahl der C-Atome eingegeben und durch eine switch-Anweisung ausgewertet werden können, sondern auch die Stoffklasse, codiert durch eine Zahl zwischen 1 und 6:

1 = Alkane, 2 = Alkene, 3 = Alkine, 4 = Alkohole, 5 = Aldehyde oder 6 = Alkansäuren.

Wird beispielsweise die C-Atom-Anzahl 4 sowie die Stoffklasse 5 eingegeben, so soll die Verbindung "Butanal" ausgegeben werden. Bei C = 6 und Stoffklasse 4 soll "Hexanol" ausgegeben werden, bei C = 7 und Stoffklasse 6 soll "Heptansäure" ausgegeben werden und so weiter. Die Stoffklasse soll ebenfalls durch eine switch-Anweisung ausgewertet werden.

Übung 3.10-1 (anspruchsvollere Variante)

Wie die einfache Variante, allerdings sollen die Stoffklassen im Klartext (als String) eingegeben werden können: "Alkan", "Alken", "Alkin", "Alkohol", "Aldehyd" oder "Alkansäure". Die Auswertung durch eine switch-Anweisung ist dann bei den Stoffklassen nicht möglich, hier müssen Sie geschachtelte if-Anweisungen einsetzen.

Bei den Stoffklassen sind außerdem folgende Synonyme zu berücksichtigen:

"Alkanol" = "Alkohol"

"Alkanal" = "Aldehyd"

"Carbonsäure" = "Alkansäure"

Übung 3.10-2

Alkane mit einer OH-Gruppe irgendwo im Molekül werden als Alkohole oder Alkanole bezeichnet. Dabei kommt es auf die Stellung der OH-Gruppe an. Ein Butan mit einer OH-Gruppe am ersten C-Atom wird als "Butan-1-ol" bezeichnet. Befindet sich die OH-Gruppe am zweiten C-Atom, spricht man von einem "Butan-2-ol". Wenn die OH-Gruppe am dritten C-Atom ist, liegt kein "Butan-3-ol" vor, sondern ebenfalls ein "Butan-2-ol". Man könnte ja das "Butan-3-ol" einfach um 180 Grad drehen, und dann wäre das dritte C-Atom wieder das zweite C-Atom. Entsprechend gibt es auch kein "Butan-4-ol", das wäre ja nichts anderes als ein "Butan-1-ol".

Schreiben Sie eine Klasse **Alkohol** mit einer Methode

```
public String gibAlkohol(int c, int pos)
```

Dabei ist c die Zahl der C-Atome und pos die Position der OH-Gruppe. Die Methode soll den korrekten Namen des Alkohols zurück geben, also beispielsweise "Pentan-3-ol".

Folge 4 - Schleifen

Unterrichtsvorhaben

4.1 Entwurf einer Klasse "Auto" (I1,I2,I3 / A,I,D)

Schritt 1 - neues Projekt erstellen

Erzeugen Sie eine neues leeres Projekt mit der Klasse **Auto**.

Doppelklicken Sie auf die Klasse **Auto** um den Quelltext zu bearbeiten. Bereinigen Sie dann den Quelltext so, dass nur noch der Minimalquelltext übrig ist:

```
public class Auto
{
   public Auto()
   {
   }
}
```

Schritt 2 - das Auto kann fahren

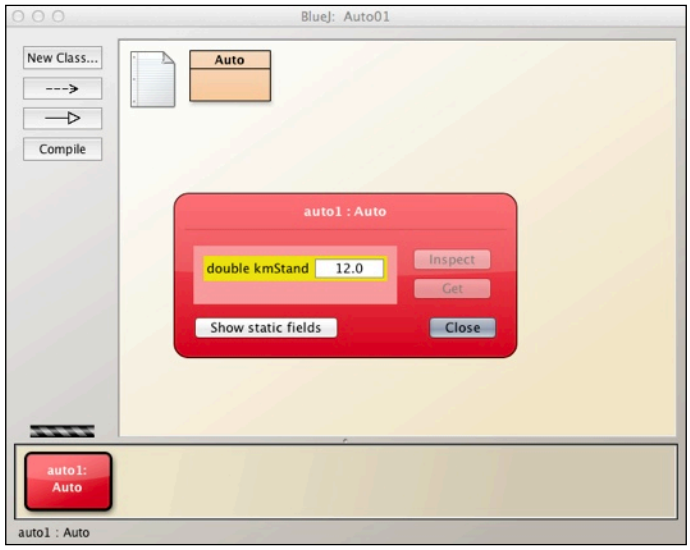
Was soll unser Auto alles können? Ähnlich wie bei dem Waage-Projekt wollen wir uns zunächst Gedanken darüber machen, welche Methoden die Auto-Objekte benötigen, damit sie "fahren" können.

```
public void fahren(double km)
```

Das wäre ein erster Vorschlag für eine Methode zum Fahren. Wir geben beispielsweise den Wert 12.0 als Parameter an, und unser Auto legt eine Strecke von 12.0 km zurück. Natürlich muss diese Streckenveränderung irgendwo gespeichert werden, und damit kommen wir auch schon zum ersten Attribut unserer Klasse.

```
public class Auto
{
   double kmStand;

   public Auto()
   {
      kmStand = 0;
   }

   public void fahren(double km)
   {
      kmStand += km;
   }
}
```

Geben Sie diesen Quelltext ein, kompilieren Sie und erzeugen Sie ein Objekt **auto1** der Klasse **Auto**. Rufen Sie dann die Methode **fahren()** mit dem Wert 12.0 auf und inspizieren Sie das Objekt.



4.1 - 1 Wir inspizieren nach dem Fahren das Objekt

Das Attribut **kmStand** hat tatsächlich den Wert 12.0

Der += Operator und seine Verwandten

Im Quelltext der Methode **fahren()** hätte man natürlich auch schreiben können:

```
kmStand = kmStand + km;
```

Eine Zuweisung, die den Wert einer Variable um eine bestimmte Zahl erhöht, nennt man **Inkrementation**. Eine solche Inkrementation kann man auch einfacher schreiben:

```
kmStand += km;
```

Diese kurze Anweisung mit dem `+=` Operator besagt: Erhöhe den Wert der Variable **kmStand** um den Wert **km**.

Neben dem `+=` Operator gibt es auch den `-=`, den `*=` und den `/=` Operator.

Übungen

Übung 4.1-1 (6 Punkte)

Ergänzen Sie den Quelltext der Klasse **Auto** um ein Attribut, das die noch vorhandene Benzinmenge in Litern speichert. Der Tank des Autos soll maximal 70 Liter Benzin fassen, und das Auto soll durchschnittlich 7.3 Liter pro 100 km Fahrstrecke verbrauchen. Verändern Sie nun die Methode **fahren()** derart, dass beim Fahren tatsächlich Benzin verbraucht wird. Der Kilometerstand und der Benzinstand des Autos sollen durch eine weitere neue Methode **anzeigen()** in der Konsole ausgegeben werden. Schreiben Sie auch diese Methode.

Übung 4.1-2 (2 Punkte)

Machen Sie Ihr Auto flexibler. Sowohl das Fassungsvermögen des Benzintanks wie auch der durchschnittliche Verbrauch sollen in neuen Attributen gespeichert werden, deren Werte im Konstruktor dann mithilfe von Parametern gesetzt werden:

```
public Auto(double kmStand, double tankvolumen, double verbrauch)
```

Beim Aufruf des Konstruktors können Sie gleich bei der Erzeugung neuer **Auto**-Objekte die entsprechenden Attributwerte festlegen. Verschiedene Objekte der Klasse **Auto** können auf diese Weise unterschiedliche Attributwerte haben.

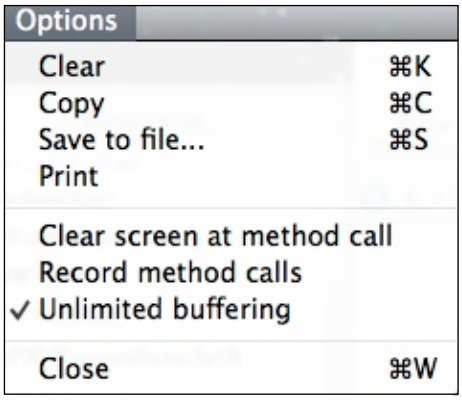
Exkurs - Formatierte Ausgabe (fakultativ)

Betrachten Sie die folgende **anzeigen()**-Methode:

```
public void anzeigen()
{
    System.out.printf("Kilometerstand = %10.2f\n",kmStand);
}
```

Hier wird nicht der **println()**-Befehl verwendet, sondern der **printf()**-Befehl. Das "f" steht für **formatierte Ausgabe**. Durch den Formatstring `"%10.2f"` wird erreicht, dass die nachfolgende double- oder float-Zahl mit exakt zehn Stellen, davon zwei **Nachkommastellen** ausgegeben wird, was die Konsolenausgabe doch sehr viel ansprechender macht. Der **Formatierungsbefehl** `"\n"` sorgt für einen **Zeilenvorschub**, entspricht also dem `"ln"` beim **println()**-Befehl.

Angenommen, wir haben ein Auto mit einem Kilometerstand von 1500 km. Dann rufen wir **anzeigen()** auf. Nun rufen wir **fahren(5000)** und wieder **anzeigen()** auf, anschließend **fahren(6000)** und noch einmal **anzeigen()**. Bei den Konsolen-Einstellungen müssen wir allerdings vorher dafür sorgen, dass die Konsole beim Aufruf einer neuen Methode nicht gelöscht wird.



***4.1 - 2** Das Konsolen-Menü von BlueJ*

Der Eintrag "Clear screen at method call" darf nicht aktiviert sein. Hier das Ergebnis unserer Bemühungen in der Konsole:

```
Kilometerstand =     1500,00
Kilometerstand =     6500,00
Kilometerstand =    12500,00
```

***4.1 - 3** Die Ausgabe in der Konsole*

System.out.printf()

Dieser Befehl dient zur formatierten Ausgabe in die Textkonsole. In den auszugebenden Text können spezielle Formatstrings eingebaut werden, die dann bei der Ausgabe durch konkrete Zahlen ersetzt werden.

Schritt 3 - Testklassen (fakultativ)

Es ist Ihnen sicherlich auch schon aufgefallen, dass man beim Entwickeln einer Klasse diese recht häufig testen muss. Sobald man den Quelltext etwas verändert hat, muss man neu kompilieren, und alle Objekte, die man vorher angelegt hatte, sind wieder verschwunden. Also muss wieder ein neues Objekt erstellt werden, und alle Methoden müssen wieder aufgerufen werden. Oft haben diese Methoden mehrere Parameter, die dann auch noch angegeben werden müssen.

All diese Arbeit kann man sich erleichtern, wenn man die Klasse - hier **Auto** - von einer Testklasse prüfen lässt. Hier der Quelltext einer Klasse **Test**, welche die Klasse **Auto** überprüft:

```
public class Test
{
    Auto otto;

    public Test()
    {
       otto = new Auto(10000,70,7.5);

       otto.fahren(20);
       otto.anzeigen();
       otto.fahren(60);
       otto.anzeigen();
       otto.fahren(20);
       otto.anzeigen();
       otto.fahren(30);
       otto.anzeigen();
       otto.fahren(80);
       otto.anzeigen();
       otto.fahren(280);
       otto.anzeigen();
    }
}
```

Zuerst wird im Konstruktor der Testklasse (andere Methoden sind hier gar nicht nötig) ein Auto-Objekt **otto** (Ottomotor) initialisiert, und zwar ein Auto, das schon 10.000 km gefahren ist, ein Tankvolumen von 70 Litern hat und einen durchschnittlichen Verbrauch von 7.5 Litern / 100 km hat. Anschließend fährt das Auto 20 km, danach werden die wichtigsten Attributwerte angezeigt, zum Beispiel der aktuelle Benzinverbrauch, der ja von der gefahrenen Strecke abhängt. Danach wird die **fahren()**-Methode mehrmals mit verschiedenen Werten aufgerufen, und jedes Mal werden die Attributwerte durch Ausführen der **anzeigen()**-Methode überprüft.

Mithilfe dieser Testklasse und der Konsolenausgabe lässt sich leicht überprüfen, ob der Quelltext der zu entwickelnden Klasse wirklich das tut, was er soll. Und es reicht, wenn man nach jeder Veränderung der zu überprüfenden Klasse das Projekt kompiliert und dann ein Objekt der Testklasse erzeugt. Da der gesamte Test bereits im Konstruktor der Testklasse durchgeführt wird, muss nach dem Erzeugen des Testklassen-Objekts auch keine weitere Methode mehr aufgerufen werden, um den eigentlichen Test durchzuführen.

Schritt 4 - Tanken (obligatorisch)

Übung 4.1-3 (4 Punkte)

Erweitern Sie Ihre Klasse **Auto** so, dass das Auto wieder vollgetankt werden kann. Ein Attribut für das Tankvolumen hatten Sie ja bereits in der Übung 4.1-2 angelegt; wenn nicht, müssen Sie das jetzt nachholen.

Ein Auto tankt man natürlich, bevor der Tank ganz leer ist. Schreiben Sie die **tanken()**-Methode so, dass auch dann getankt werden kann, wenn noch Benzin im Tank enthalten ist. Auch hier soll das Tankvolumen nicht überschritten werden können.

Schritt 5 - Fahren (obligatorisch)

Wir wollen jetzt das Auto "richtig" fahren lassen. Den Graphikmodus von Java beherrschen wir noch nicht, darum müssen wir das Fahren in der Textkonsole simulieren.

Wir hatten ja bereits eine Methode zum Fahren programmiert, die zum Beispiel so aussehen könnte:

```
public void fahren(double km)
{
   if (tankinhalt > km * verbrauch/100)
   {
      kmStand += km;
      tankinhalt -= km * verbrauch/100;
   }
   else
   {
      kmStand += tankinhalt / (verbrauch/100);
      tankinhalt = 0;
   }
}
```

Dies ist eine mögliche Version der **fahren()**-Methode, der Kilometerstand wird um die gefahrenen Kilometer erhöht, der Tankinhalt wird verringert, allerdings ist der Benzinverbrauch hier konstant und nicht von der Außentemperatur oder der Motortemperatur abhängig. Hier würde sich ein Betätigungsfeld für unterforderte Schüler(innen) bieten...

Übung 4.1-4 (2 Punkte)

Schreiben Sie eine Testklasse, die ein Objekt der Klasse **Auto** 50 mal 1 km fahren lässt.

Machen Sie hier kräftig Gebrauch von Copy & Paste; sie wollen doch wohl nicht jede Zeile einzeln eintippen?

Schritt 6 - Schleifen (obligatorisch)

Hier sehen Sie die ersten Zeilen der Testklasse, die ein Auto 50 km fahren lässt:

```
public class Test
{
    Auto otto;

    public Test()
    {
       otto = new Auto(10000,70,7.5);
       otto.fahren(1); otto.anzeigen();
       otto.fahren(1); otto.anzeigen();
       otto.fahren(1); otto.anzeigen();
       otto.fahren(1); otto.anzeigen();
       otto.fahren(1); otto.anzeigen();
       otto.fahren(1); otto.anzeigen();
       otto.fahren(1); otto.anzeigen();
       ...
```

Auf die restlichen 43 gleichen Zeilen habe ich hier aus verständlichen Gründen verzichtet.

Betrachten Sie nun folgende Version der Testklasse, die das Gleiche leistet, aber sehr viel kürzer ist:

```
public class Test
{
    Auto otto;

    public Test()
    {
       otto = new Auto(10000,70,7.5);
       otto.anzeigen();

       int strecke = 0;
       while (strecke < 50)
       {
           otto.fahren(1);
           otto.anzeigen();
           strecke++;
       }
    }
}
```

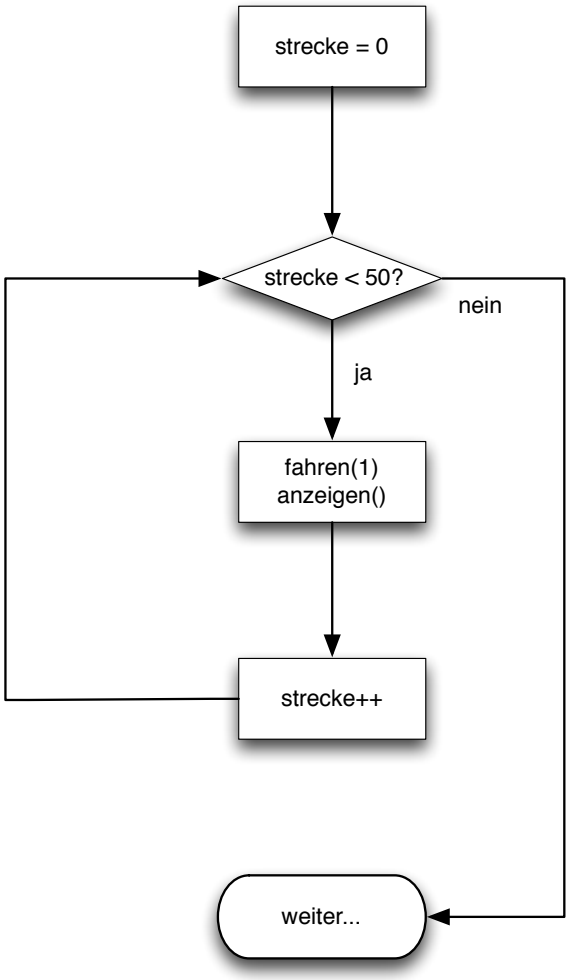
Der wichtigste Teil dieser Methode ist sicherlich die **while-Schleife**, die im Quelltext farbig hinterlegt wurde.

Mit dem Autofahren sind wir jetzt eigentlich am Ende. Wenn Sie wollen, können Sie ja ihre Klasse **Auto** noch genauer modellieren und zum Fahren dann eine while-Schleife benutzen. Im folgenden Abschnitt wollen wir auf das Thema while-Schleifen noch etwas genauer eingehen; vor allem die Klausur-Leute sollten sich diesen Abschnitt eingehend anschauen.

4.2 while-Schleifen (I1,I2,I3 / A,M,I,D)

Programmfluss:

Der Ablauf der einzelnen Befehle und Überprüfungen in einer while-Schleife wird am besten in Form eines **Flussdiagramms** dargestellt:



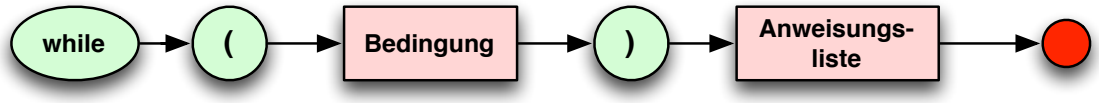
***4.2-1** Flussdiagramm einer while-Schleife*

In dem Flussdiagramm fällt sofort ein typisches Merkmal von while-Schleifen auf: Die Abfrage, ob die Schleife (ein weiteres Mal) durchlaufen werden soll, findet *am Anfang* der Schleife statt. Wird die Frage `strecke < 50)` mit `true` beantwortet, so wird die Schleife durchlaufen, andernfalls nicht. Nach dem Durchlaufen der while-Schleife oder nachdem diese übersprungen wurde (falls die Schleifenbedingung nicht erfüllt wurde), wird mit dem Java-Befehl weitergemacht, der nach der while-Schleife kommt.

Während ein **Flussdiagramm** den **Programmfluss** zeigt (welche Befehle werden in welcher Reihenfolge ausgeführt?), zeigt ein **Syntaxdiagramm** den "grammatikalischen" Aufbau eines Befehls, in unserem Fall also den Aufbau einer while-Schleife.

Syntax:

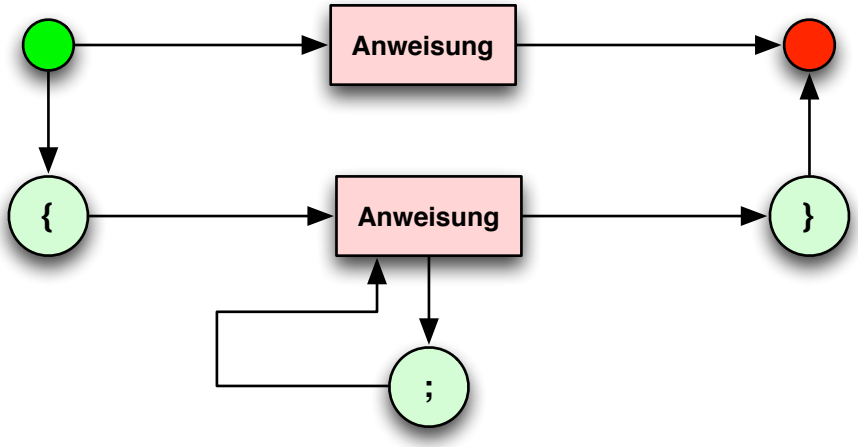
Hier zunächst ein **Syntaxdiagramm** für die while-Schleife:



4.2-2 Syntaxdiagramm einer while-Schleife bzw. while-Anweisung

Dieses Syntaxdiagramm kann man nun folgendermaßen lesen:

Eine while-Schleife besteht aus dem **Schlüsselwort** `while`, gefolgt von einer in Klammern stehenden Bedingung, die entweder `true` oder `false` ist. Dann kommt eine Anweisungsliste:



4.2-3 Syntaxdiagramm einer Anweisungsliste

Eine **Anweisungsliste** ist entweder eine einzelne **Anweisung** oder eine Liste von Anweisungen, die durch geschweifte Klammern zusammengehalten wird. Die einzelnen Anweisungen werden durch Strichpunkte (Semikola) getrennt:

Was man noch über while-Schleifen wissen sollte:

Typisch für eine while-Schleife ist, dass die logische Bedingung überprüft wird, *bevor* der Schleifenrumpf (also die Anweisungsliste) durchlaufen wird. Man spricht hier auch von einer **vorprüfenden Schleife**. Wenn die logische Bedingung vor dem ersten Schleifendurchlauf *nicht* erfüllt ist, dann wird die while-Schleife kein einziges Mal durchlaufen. Das kann durchaus mal passieren. Will man aber, dass die Schleife auf jeden Fall mindestens einmal durchlaufen wird, so muss man entweder dafür sorgen, dass die Schleifenbedingung mindestens einmal erfüllt ist, oder man ersetzt die while-Schleife durch eine sogenannte do-while-schleife (siehe den folgenden Exkurs).

Die Schleife wird abgebrochen, sobald die Bedingung nicht mehr erfüllt ist. Für einen solchen Abbruch muss man aber beim Programmieren der Schleife sorgen. Achtet man nicht darauf, führt das zu einer Endlosschleife; die Schleife wird nie beendet, das Programm stürzt in der Regel ab.

Übungen

Übung 4.2-1 (2 Punkte)

Geben Sie an, was genau durch die folgende while-Schleife berechnet wird!

```
int x = 1;
int y = 1;
while (x <= 10)
{
   y = y * x;
   x++;
}
```

Übung 4.2-2 (4 Punkte)

Betrachten Sie die folgende while-Schleife:

```
while (a != b)
{
   if (a > b) a = a-b;
   else b = b-a;
}
erg = a;
```

Geben Sie an,
a) welche Zahl diese while-Schleife berechnet, wenn a=35 und b = 15 ist (2 Punkte)
b) was der Algorithmus überhaupt (allgemein) berechnet (2 Punkte)

Übung 4.2-3 (für Experten, 4 Punkte)

Die Konstante Pi hat den Wert 3.14159....

PiPi Langstrumpf hat nun folgende Idee:

"Wenn ich 10 durch 3 teile, erhalte ich 3.3333..., das ist schon ein guter Näherungswert für Pi - allerdings ist 10/3 größer als Pi. Ich erhöhe also den Nenner. Dummerweise ist 10/4 = 2.50 wieder kleiner als Pi. Also erhöhe ich jetzt den Zähler.

11/4 = 2.75 ist immer noch kleiner als Pi, daher erhöhe ich den Zähler noch einmal.
12/4 = 3.00 reicht auch noch nicht, versuchen wir es mit 13/4 = 3.25.

Jetzt ist der Quotient wieder zu groß, daher erhöhe ich den Zähler und erhalte 13/5 = 2.6. Dann muss ich wieder den Nenner erhöhen. 14/5 = 2,8; 15/5 = 3.0; 16/5 = 3.2 - wieder ist der Quotient größer als Pi. Also muss jetzt wieder der Nenner erhöht werden und so weiter.

Wenn ich das oft genug wiederhole, erhalte ich schließlich einen Nenner n und einen Zähler z, deren Quotient z/n den Wert 3.14159 liefert."

Schreiben Sie eine Java-Klasse **BerechnePi**, die ähnlich vorgeht, wie PiPi das beschrieben hat. Setzen Sie dazu eine while-Schleife ein, die erst dann abbricht, wenn die Differenz zwischen z/n und 3.14159 kleiner als 0.00001 ist. Lassen Sie jeden Zwischenschritt ausgeben, ungefähr so:

```
10.0/4.0=2.5
11.0/4.0=2.75
12.0/4.0=3.0
13.0/4.0=3.25
13.0/5.0=2.6
14.0/5.0=2.8
15.0/5.0=3.0
16.0/5.0=3.2
16.0/6.0=2.6666666666666665
17.0/6.0=2.8333333333333335
18.0/6.0=3.0
19.0/6.0=3.1666666666666665
19.0/7.0=2.7142857142857144
20.0/7.0=2.857142857142857
21.0/7.0=3.0

...

352.0/112.0=3.142857142857143
352.0/113.0=3.1150442477876106
353.0/113.0=3.1238938053097347
354.0/113.0=3.1327433628318584
355.0/113.0=3.1415929203539825
```

4.3 do-while-Schleifen (I1,I2,I3 / A,M,I,D)

While-Schleifen sind **vorprüfende** Schleifen. *Vor* jedem Schleifendurchgang wird die Schleifenbedingung überprüft, und wenn sie nicht erfüllt ist, wird die Schleife erst gar nicht durchlaufen. Es kann daher also der Fall eintreten, dass eine while-Schleife kein einziges Mal durchlaufen wird.

```
int x = 10;
while (x > 20)
   x--;
System.out.println(x);
```

Hier haben wir ein Beispiel für eine solche Schleife. Da 10 nicht größer ist als 20, wird die Schleife kein einziges Mal durchlaufen, am Ende des Quelltextes wird der unveränderte Wert 10 ausgegeben.

Wenn man möchte, dass eine Schleife auf jeden Fall mindestens einmal durchlaufen wird, muss man entweder dafür sorgen, dass die Schleifenbedingung mindestens einmal erfüllt ist, oder man ersetzt die while-Schleife durch eine do-while-Schleife. Hier ein Beispiel

```
int x = 10;
do
   x--;
while (x > 20);
System.out.println(x);
```

Da die Schleifenbedingung erst *am Ende* der Schleife überprüft wird, wenn die Schleife also bereits einmal durchlaufen wurde, wird die Zahl 9 ausgegeben, denn der Befehl `x--` wird auf jeden Fall einmal ausgeführt, egal, ob die Schleifenbedingung erfüllt ist oder nicht. Do-while-Schleifen sind also **nachprüfende** Schleifen bzw. **nicht-abweisende** Schleifen, wie man auch am Flussdiagramm in der Abbildung 4.4-1 erkennen kann. Irgendwo habe ich auch mal den Begriff "**fußgesteuert**" gefunden, der do-while-Schleifen gut kennzeichnet. Entsprechend müsste man dann while-Schleifen als "**kopfgesteuert**" bezeichnen, was gar nicht mal so schlecht ist.

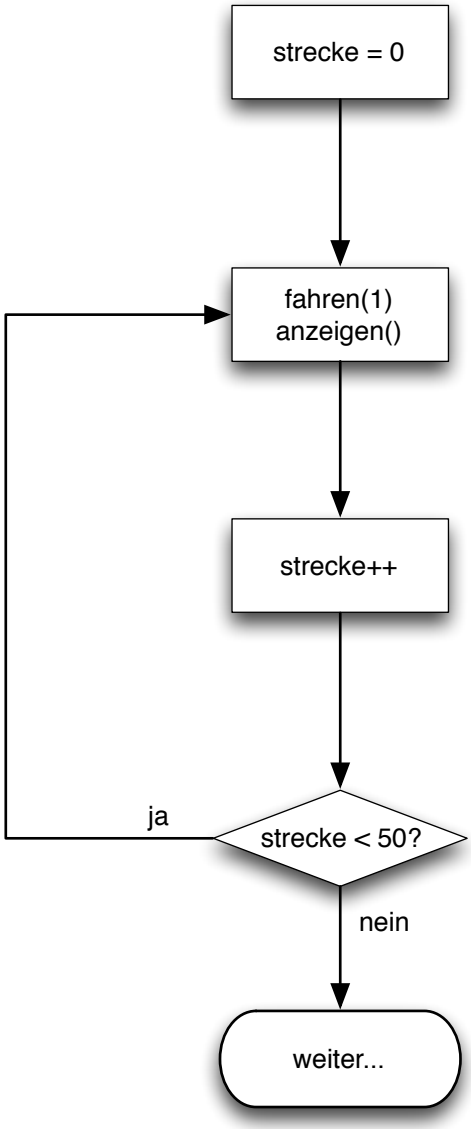
Für Pascal-Kenner: repeat-until-Schleifen

Eine do-while-Schleife darf nicht mit einer repeat-until-Schleife aus der Programmiersprache Pascal verwechselt werden. Vergleichen wir einmal die beiden folgenden Schleifen:

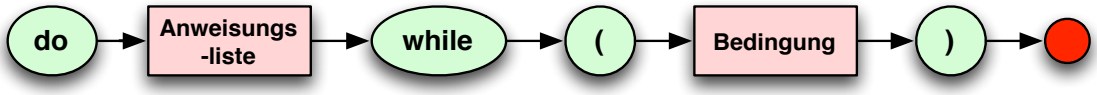
do-while-Schleife	repeat-until-Schleife

`int x = 10;`
`do`
`  x--;`
`while (x > 20);`
`System.out.println(x);`
	`x := 10;`
	`repeat`
	`  x := x-1;`
	`until x > 20;`
	`writeln(x);`

Die do-while-Schleife würde nach dem ersten Schleifendurchgang mit x = 9 abbrechen. Die repeat-until-Schleife dagegen würde endlos wiederholt werden, da die Abbruch-Bedingung x > 20 nie eintritt. Die Schleifenbedingung bei der do-while-Schleife ist nämlich keine Abbruch-Bedingung, sondern das Gegenteil davon, eine Art Fortsetzungs-Bedingung oder Weitermachen-Bedingung (mir fehlt hier der korrekte Fachbegriff...).



4.3-1 Flussdiagramm einer do-while-Schleife



4.3-2 Syntaxdiagramm einer do-while-Schleife

4.4 Vorzeitiger Abbruch von while-Schleifen (I2,I3 / I)

(fakultativ)

Manchmal kommt es vor, dass eine Schleife beendet werden muss, obwohl die Schleifenbedingung noch erfüllt ist. Diesen vorzeitigen Abbruch kann man mit dem kurzen Befehl `break` erzwingen, der in den Schleifenkörper eingebaut wird. Nicht alle Informatikerkollegen sind glücklich über den `break`-Befehl in Schleifen, dennoch werde ich ihn hier der Vollständigkeit halber erläutern.

```
while (true)
{
   i++;
   if (i >= 100) break;
   x = x+i;
}
```

Hier haben wir eine typische Anwendung des `break`-Befehls. Man konstruiert zunächst eine Endlosschleife, indem man die Schleifenbedingung `true` verwendet, die natürlich immer erfüllt ist. Sobald man dann innerhalb der Schleife sein Ziel erreicht hat (in unserem Beispiel soll `i` den Wert von 100 erreicht haben), verlässt man die Endlosschleife mit dem `break`-Befehl. Diesen Befehl darf man auch mehrmals verwenden:

```
while (true)
{
   i++;
   if (i >= 100) break;
   x = x+i;
   if (x >= 150) break;
}
```

4.5 For-Schleifen (I1,I2,I3 / A,M,I,D)

Eine weitere Art von Schleifen neben den while- und den do-while-Schleifen sind die for-Schleifen, auch **Zählschleifen** genannt. Während bei einer while-Schleife zu Beginn der Schleife nicht festgelegt ist, wo oft die Schleife durchlaufen werden soll, setzt man for-Schleifen ein, wenn man bereits beim Schreiben des Programms weiß, wie häufig die Schleife ausgeführt werden soll. Ein schönes Beispiel ist die Aufgabe, die Summe der Zahlen 1, 2, 3, ..., 100 zu bilden. Natürlich kann man hierfür eine while-Schleife einsetzen, aber mit einer for-Schleife geht das einfacher:

while-Schleife	for-Schleife

`int sum = 0;`
`int i = 1;`
`while (i <= 100)`
`{`
`  sum += i;`
`  i++;`
`}`
	`int sum = 0;`
	
	`for (int i=1; i <= 100; i++)`
	`  sum += i;`

4.5.1 Der Schleifenkopf

Der **Schleifenkopf** einer for-Schleife besteht aus drei Elementen. Zunächst wird die **Laufvariable** initialisiert, eventuell sogar deklariert:

```
int i = 1
```

In diesem Beispiel wird die Laufvariable `i` zunächst deklariert (`int i`) und dabei gleichzeitig mit einem **Startwert** initialisiert (`i = 1`). Die Laufvariable muss natürlich nicht immer `i` heißen, aber irgendwie hat es sich eingebürgert, die meisten Laufvariablen so zu bezeichnen.

Das zweite Element im Schleifenkopf ist die **Schleifenbedingung**. Genau wie bei einer while-Schleife wird die for-Schleife nur dann durchlaufen, wenn die Schleifenbedingung erfüllt ist:

```
i <= 100
```

Wenn die Laufvariable den Wert `100` nicht überschritten hat, wird die Schleife durchlaufen.

Das dritte Element des Schleifenkopfs schließlich ist die **Inkrementation** oder **Dekrementation** der Laufvariable:

```
i++
```

In diesem Beispiel wird die Laufvariable `i` um den Wert `1` erhöht, aber erst dann, wenn der Schleifenkörper vollständig durchlaufen wurde.

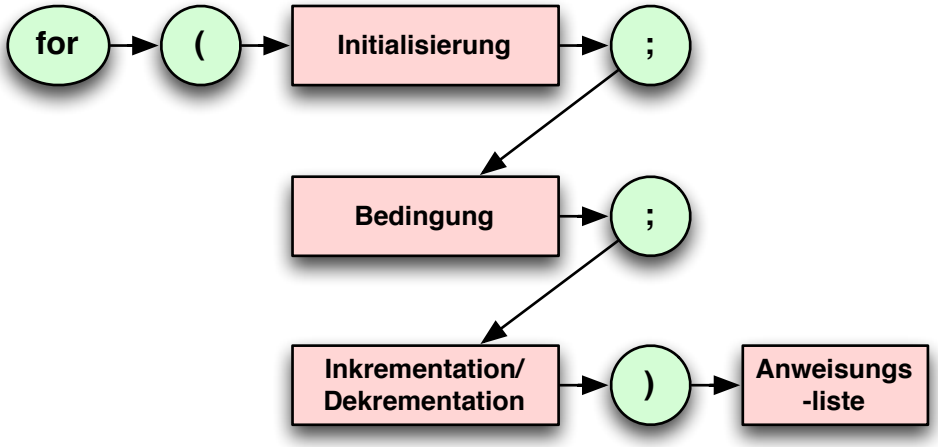
Übung 4.5-1 (2 Punkte)

Schreiben Sie eine Methode

```
public int fakultaet(int n)
```

welche die Fakultät einer natürlichen Zahn n berechnet und als int-Wert zurück liefert. Die Fakultät von n=5 ist beispielsweise 1 * 2 * 3 * 4 * 5 = 120.

4.5.2 Geschachtelte for-Schleifen



***4.5-1** Syntaxdiagramm einer for-Schleife*

Wie man dem Syntaxdiagramm einer for-Schleife gut entnehmen kann, besteht der Schleifenkörper aus einer **Anweisungsliste**. Bereits bei der if-else-Anweisung hatten wir über Anweisungslisten gesprochen. Eine Anweisungsliste ist

- entweder **leer**
- besteht aus einer einzigen **Anweisung** oder
- besteht aus einer **Liste von Anweisungen**, die durch geschweifte Klammern eingeschlossen sind. Nach jeder Anweisung folgt ein Semikolon.

Eine Anweisung kann nun entweder eine **einfache Anweisung** wie z.B. eine Zuweisung oder ein Methodenaufruf sein, aber auch eine **if-else-Anweisung**, eine **while-Schleife**, eine **do-while-Schleife** oder eine weitere **for-Schleife**. Dies ermöglicht die Konstruktion verschachtelter Schleifen. In einer "äußeren" Schleife kann eine "innere" Schleife enthalten sein. Auch drei oder vier Schachtelungsebenen sind denkbar (wenn auch vielleicht nicht allzu sinnvoll).

```
for (int k = 20; k >= 2; k = k-2)
{
   for (int j = 1; j <= k; j++) otto.vor();
   otto.linksUm();
}
```

Das Objekt **otto** würde in der inneren for-Schleife (rot gedruckt) eine bestimmte Zahl `k` von Schritten vorwärts gehen. Wie groß die Zahl der Schritte `k` jeweils ist, wird durch die äußere for-Schleife vorgegeben. Wenn **otto** nach vorn gegangen ist, dreht er sich nach links. Dann ist der Schleifenkörper der äußeren for-Schleife einmal durchlaufen, und `k` wird um den Wert 2 dekrementiert. Ist **otto** also eben noch 20 Schritte nach vorne gegangen, wird er beim nächsten Mal nur noch 18 Schritte nach vorne gehen und sich dann wieder nach links drehen.

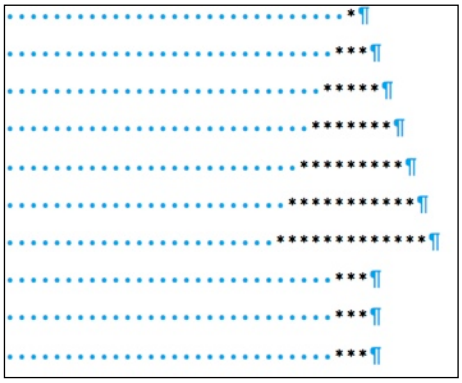
Übungen

Übung 4.5-2 (2 Punkte)

Was macht die folgende for-Schleife ?

```
for (int a = 1; a <= 10; a++)
{
   for (int b = 1; b <= 10; b++)
      System.out.printf("%6d", a*b);
   System.out.println();
}
```

Wir zeichnen einen Baum



***4.5-2** Ein Baum in der Textkonsole. Leerzeichen und Zeilenumbrüche sind blau markiert.*

Betrachten Sie die Abbildung oben. Das Bild wurde mit folgender Java-Methode gezeichnet:

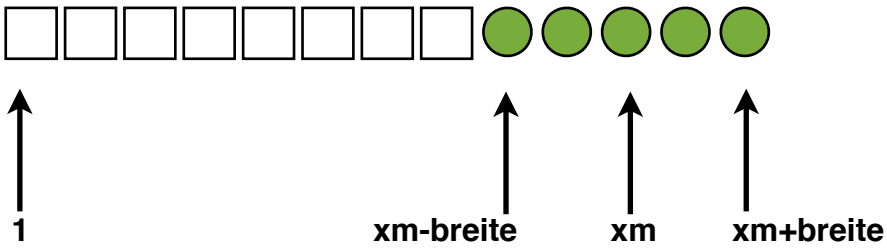
```
public void zeichneBaum(int maxBreite, int xm)
{
    zeichneAst(0,xm);
    zeichneAst(1,xm);
    zeichneAst(2,xm);
    zeichneAst(3,xm);
    zeichneAst(4,xm);
    zeichneAst(5,xm);
    zeichneAst(6,xm);
    zeichneAst(1,xm);
    zeichneAst(1,xm);
    zeichneAst(1,xm);
 }
```

Der Parameter `maxBreite` wird hier noch gar nicht benutzt, könnte also in dieser Version der Methode auch weggelassen werden. Der Parameter `xm` hatte bei der Erzeugung der Ausgabe (Abb. 4.5-2) den Wert `30`.

Die Methode **zeichneAst()**, die von der Methode **zeichneBaum()** aufgerufen wird, sieht so aus:

```
public void zeichneAst(int breite, int xm)
{
    for (int x=1; x<xm-breite; x++)
        System.out.print(" ");
    for (int x=xm-breite; x<=xm+breite; x++)
        System.out.print("*");
    System.out.println();
 }
```

Grundidee bei der Methode zeichneAst()



***4.5-3** Visualisierung der Grundidee*

Der Wert `xm` bezeichnet die Spalte in der Textkonsole, in der sich der Stamm des Baumes befinden soll. Für den Baum in Abbildung 4.5-2 wurde `xm=30` verwendet; in der Zeichnung 4.5-3 hat `xm` den Wert 11. Der Wert `breite` ist die Breite eines Astes. Vom Stamm gehen immer zwei gleich breite Äste aus. In der Abbildung oben hat `breite` den Wert `2`.

Zunächst müssen Leerzeichen "gedruckt" werden, also mit dem **System.out.printf()**-Befehl ausgegeben werden. Dafür ist die erste for-Schleife verantwortlich. Die Anzahl der Leerzeichen ergibt sich aus dem `xm`-Wert und der Breite des Astes (siehe Quelltext). Nach Beendigung dieser ersten for-Schleife darf auf gar keinen Fall ein Zeilenumbruch ausgeführt werden; der Textcursor muss am Ende der Leerzeichen stehen bleiben.

Nun werden die Sternchen ausgegeben, und zwar erst für den linken Ast, dann eines für den Stamm, und dann für den rechten Ast. Dafür ist die zweite for-Schleife zuständig. Nun muss (!) ein Zeilenumbruch kommen, damit die nächste Ebene des Baumes gezeichnet werden kann.

Übung 4.5-3 (3 Punkte)

Die **zeichneBaum()**-Methode ist noch zu wenig flexibel. Bauen Sie eine Schleife in diese Methode ein, so dass der Parameter `maxBreite` ausgewertet werden kann. Dieser Parameter gibt die Breite der beiden unteren Äste an (2 Punkte). Außerdem soll die Länge des Stammes mit `maxBreite` übereinstimmen. Ist `maxBreite = 7`, soll auch der Stamm aus 7 Ebenen bestehen (1 Punkt).

Übung 4.5-4 (3 Punkte)

Falls Sommer: Der Baum soll nun mit Äpfeln versehen werden.
Falls Winter: Der Baum soll nun mit Weihnachtskugeln geschmückt werden.

Dazu eignet sich der Buchstabe 'o' sehr gut. Verändern Sie die **zeichneAst()**-Methode derart, dass jeder zweite oder dritte Stern durch einen Apfel / eine Kugel ersetzt wird.

Ergänzungsübung 4.5-5 (für Experten; 3 Punkte)

Die Äpfel/Kugeln sind viel zu regelmäßig angeordnet. Lassen Sie sich etwas einfallen, damit sie natürlicher (zufälliger) verteilt sind.

Übung 4.5-6 (2, 3 oder 4 Punkte)

Schreiben Sie eine Klasse **QFunktion** (für quadratische Funktion), in deren Attributen die Koeffizienten a, b und c einer quadratischen Funktion `f(x) = ax² + bx + c` gespeichert sind.

Der Konstruktor soll diese drei Werte in Form von drei double-Parametern erhalten und dann in den Attributen speichern.

Schreiben Sie dann eine Ausgabe-Methode, welche die Funktion anzeigt. Hier gibt es drei unterschiedlich schwere Varianten. Angenommen, die drei Parameter haben die Werte 2, 0 und -3, dann könnten die Ausgaben so aussehen:

```
f(x) = 2.0x2 + 0.0x + -3.0
```

oder so:

```
f(x) = 2.0x2 + 0.0x -3.0
```

oder gar so:

```
f(x) = 2.0x2 - 3.0
```

Für die erste Variante bekommen Sie 2 Punkte, für die zweite 3, und für die dritte sogar 4!

Übung 4.5-7 (4 Punkte)

Erweitern Sie die Klasse **QFunktion** um eine neue Ausgabe-Methode,

```
public void ausgabeWerteTabelle(double sw)
```

welche eine Wertetabelle im Bereich x = -10 bis x = +10 erzeugt:

```
-10.0 / 98.0
-9.5 / 88.25
-9.0 / 79.0
-8.5 / 70.25
```

... und so weiter

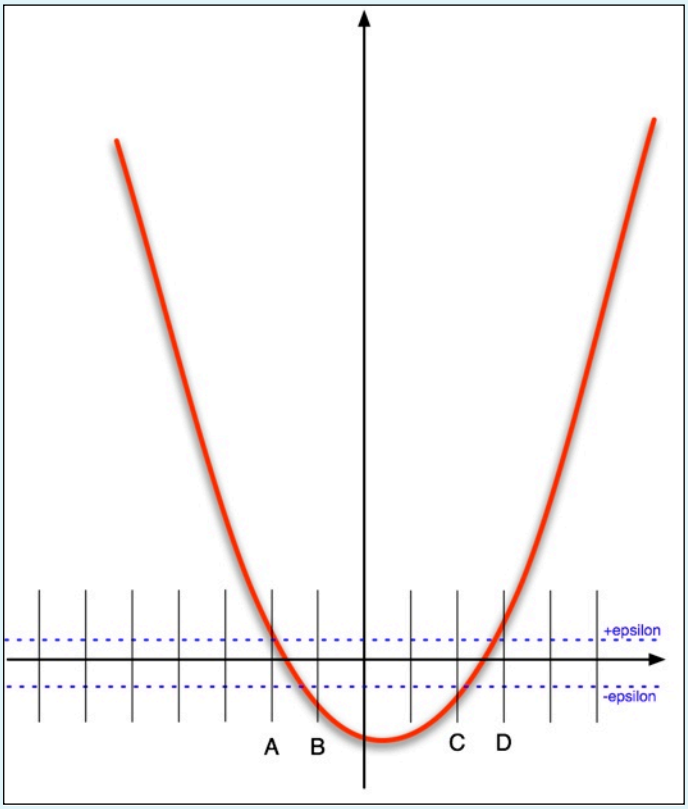
Die Schrittweite soll über einen Parameter dieser Methode eingegeben werden können. In dem Beispiel hier wurde als Schrittweite 0.5 gewählt.

3 Punkte bekommen Sie, wenn Sie die Wertetabelle besser darstellen als in diesem Beispiel (korrekte Einrückungen, Dezimalpunkte untereinander etc.). Dazu ist vielleicht eine kleine Recherche zum Thema "formatierte Ausgabe" notwendig.

Einen weiteren Punkt erhalten Sie, wenn Sie die Ausgabe flexibel gestalten, also nicht nur die Schrittweite als Parameter übergeben können, sondern auch die untere und obere Grenze des Definitionsbereichs (zum Beispiel von x = -25 bis x = +50).

Übung 4.5-8 (für Experten; 5 Punkte, evtl. 7 Punkte)

Die folgende Abbildung zeigt das Grundprinzip einer einfachen Nullstellen-Bestimmung:



In einer Schleife geht man die x-Achse systematisch ab (kleine senkrechte schwarze Striche) und ermittelt die jeweiligen y-Werte. Als "Nullstelle" gilt ein y-Wert, der innerhalb des Intervalls liegt, das durch +epsilon und -epsilon bestimmt wird (beispielsweise epsilon = 0.01).

Bei x=A liegt y beispielsweise oberhalb von +epsilon und gilt daher noch nicht als Nullstelle. Beim nächsten Schritt, x=B, liegt y aber schon unterhalb von -epsilon und gilt daher auch nicht als Nullstelle. Das gleiche Problem haben wir bei x=C und x=D.

Wenn man allerdings die Schritte auf der x-Achse kleiner macht und mit den epsilon-Werten etwas flexibler ist, kann man auf diese Weise ungefähr (!) die Lage der Nullstellen einer Funktion zweiten Grades bestimmen, falls solche Nullstellen im angegebenen Definitionsbereich überhaupt existieren.

Schreiben Sie für die Klasse **QFunktion** eine entsprechende Methode, die auf die oben beschriebene Art und Weise die Nullstellen der Funktion bestimmt und ausgibt bzw. das Nichtvorhandensein der ersten bzw. zweiten Nullstelle ausgibt.

Sie können bei der Schleife, welche die x-Achse „durchschreitet“, entweder eine feste Schrittweite einprogrammieren, dann könnte aber das oben beschriebene Problem entstehen, dass eine vorhandene Nullstelle nicht gefunden wird. Wenn Sie dieses Problem lösen, erhalten Sie zwei zusätzliche Punkte.

Folge 5 - Java-Applets

Einer der Gründe, warum so viele Leute mit Java arbeiten, ist sicherlich die Tatsache, dass man Java-Anwendungen in einem Browser laufen lassen kann, z.B. Safari oder Firefox. Allerdings kann man nicht *jede* Java-Anwendung in einem Browser anzeigen, sondern muss dazu ein so genanntes **Java-Applet** erstellen. Außerdem muss der Browser generell in der Lage sein, Java-Applet anzuzeigen. Dazu muss eine entsprechende Java-Laufzeitumgebung auf Ihrem Rechner installiert sein, manche Browser brauchen außerdem noch ein eigenes Java-PlugIn.

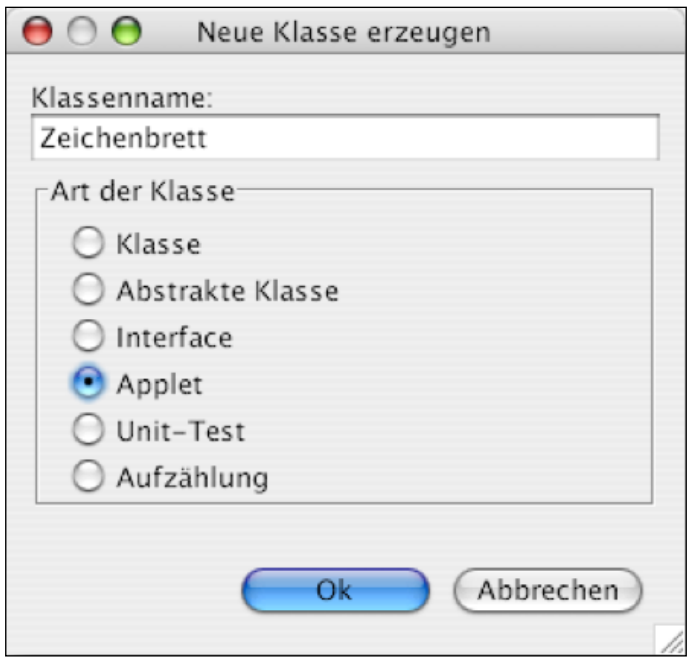
In dem ersten Unterrichtsvorhaben dieser Folge lernen Sie, wie man grundsätzlich ein Java-Applet schreibt.

Unterrichtsvorhaben 1

5.1 Tolle Graphiken zeichnen (I2,I3 / M,I,D)

Schritt 1 - Ein neues Applet erstellen

Starten Sie BlueJ und legen Sie ein ***neues Projekt*** an. Klicken Sie dann auf ***Neue Klasse*** und achten Sie darauf, dass in dem Dialogfenster ***Neue Klasse erzeugen*** der Klassentyp ***Applet*** ausgewählt ist.



5.1-1 *Ein Applet erzeugen*

In unserem Beispiel hat das Applet den Namen "Zeichenbrett".

Schritt 2 - Quelltext aufräumen

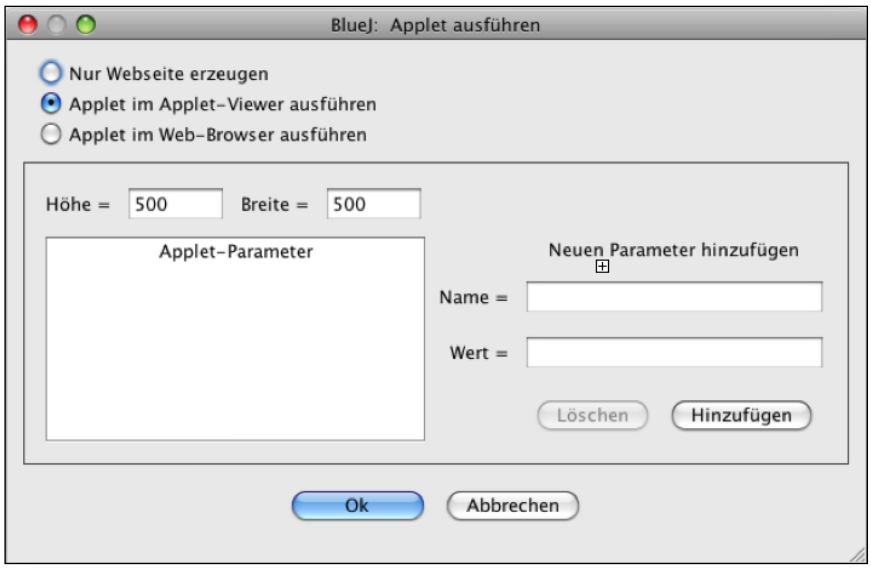
Wenn man den orangefarbenen Kasten anklickt, den BlueJ zur Darstellung der Applet-Klasse erzeugt hat, bekommt man zuerst einen kleinen Schock - so viel Quelltext! Das meiste davon ist jedoch - zumindest für unsere Zwecke - unnötig. Am besten löschen Sie den Quelltext komplett und ersetzen ihn durch folgenden **Minimal-Quelltext**:

```
import java.awt.*;
import javax.swing.*;

public class Zeichenbrett extends JApplet
{
    public void paint(Graphics g)
    {
    }
}
```

Die Quelltexte von Applets werde ich in diesem Skript übrigens mit der Farbe Rosa hinterlegen, damit sie leichter von anderen Java-Quelltexten zu unterscheiden sind. Neulich hat sich ein Schüler von mir beschwert, dass BlueJ die Applet-Quelltexte nicht ebenfalls mit einem rosafarbenen Hintergrund anzeigt. Ja Leute, da kann ich nun auch nichts dran ändern...

Schritt 3 - Kompilieren und Starten des Applets

Klicken Sie nun auf den ***Übersetzen***-Button von BlueJ. Das Applet müsste fehlerfrei übersetzt werden. Klicken Sie jetzt bitte mit der rechten Maustaste auf die Klasse **Zeichenbrett** und wählen Sie den Befehl ***Applet ausführen***. Es erscheint nun ein Dialogfenster:



5.1-2 *Dialogfenster zum Ausführen eines Applets*

Von den drei Optionen oben links im Dialogfenster sind eigentlich nur die beiden unteren interessant, ***Applet im Applet-Viewer ausführen*** und ***Applet im Web-Browser ausführen***.

Applet im Applet-Viewer ausführen ruft ein eigenes Programm auf, das vom Betriebssystem des Rechners gestartet wird und dann das Applet anzeigt. Dieses Verfahren hat den Vorteil, sehr schnell zu sein. Allerdings *könnte* es sein, dass das Applet nicht ganz genau so dargestellt wird, wie es später in einem Web-Browser erscheint. Was aber eigentlich gar kein Nachteil ist, da sowieso jeder Browser das Applet anders darstellt.

Applet im Web-Browser ausführen hat diesen Nachteil nicht, da das Betriebssystem jetzt den als Standard eingestellten Web-Browser startet und dort das Applet unter "realen" Bedingungen anzeigt. Falls der Browser noch nicht gestartet ist, dauert dieser Vorgang etwas länger.

Der Nachteil dieses Verfahrens: Sie kennen es vielleicht schon von der Darstellung von Webseiten, jeder Browser stellt die Webseite, die Sie aufrufen, etwas anders dar, darum ist es auch nicht einfach, Webseiten zu entwickeln, die auf allen Browsern wirklich gleich aussehen. Ähnliches gilt für Java-Applets; die exakte Darstellung kann vom gewählten Browser abhängig sein.

Meine persönliche Konsequenz aus diesen Überlegungen: Ich bevorzuge die Methode mit dem Applet-Viewer.

Beim Applet-Viewer können Sie ***Höhe*** und ***Breite*** des Applets einstellen. Standardmäßig sind 500 x 500 Pixel voreingestellt, diesen Wert wollen wir zunächst beibehalten. Klicken Sie jetzt einfach auf den ***OK***-Button des Dialogfensters.

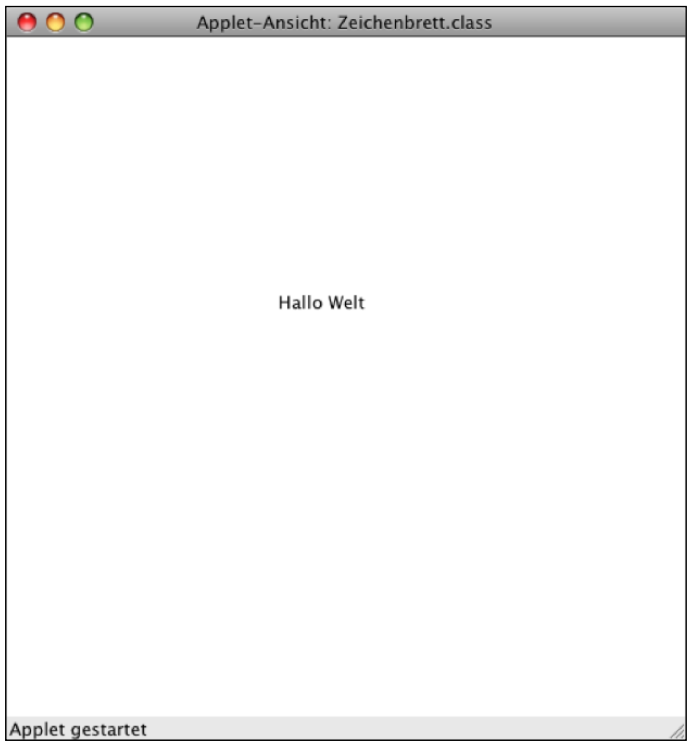
Es erscheint dann ein Fenster auf dem Bildschirm, das tatsächlich ungefähr 500 mal 500 Pixel groß ist. Leider ist dieses Applet *völlig leer*; im Applet-Viewer erscheint eine weiße Fläche, im Web-Browser entweder eine weiße oder eine graue Fläche, jenachdem welchen Browser Sie zur Verfügung haben.

Schritt 4 - "Hallo Welt"

Sie lernen jetzt den ersten Java-Zeichenbefehl kennen, nämlich **drawString()**. Verändern Sie die **paint()**-Methode des Applets folgendermaßen:

```
public void paint(Graphics g)
{
  g.drawString("Hallo Welt",200,200);
}
```

Richtig, Sie müssen nur eine einzige Zeile ergänzen. Schauen wir uns an, was diese Veränderung gebracht hat. Kompilieren Sie dazu das Applet neu und starten Sie es dann, so wie Sie es in Schritt 3 gelernt haben.



***5.1-3** Ein einfaches Hallo-Welt-Applet*

Viel sieht man ja noch nicht, aber immerhin haben wir jetzt ein erstes *nicht leeres* Applet erzeugt.

Übung 5.1-1 (3 Punkte)

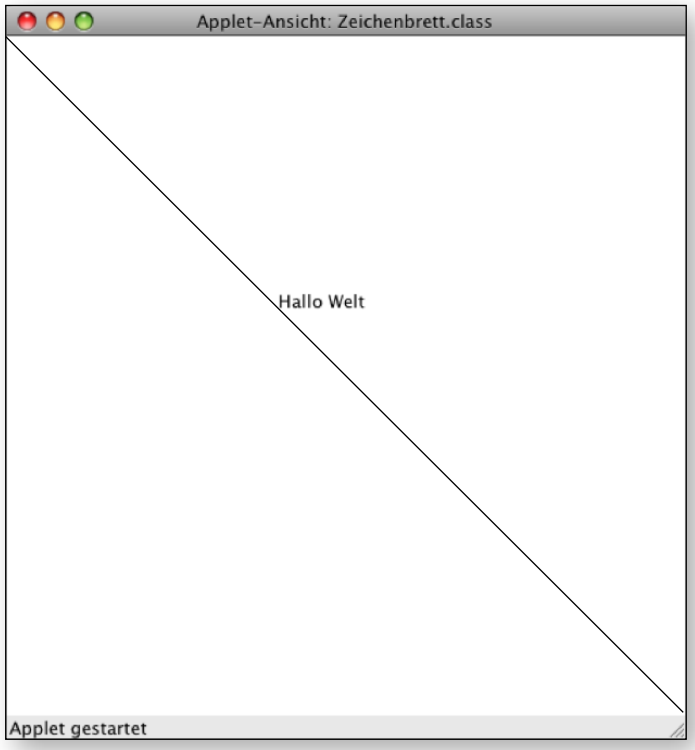
Schreiben Sie die **paint()**-Methode so um, dass der String "Hallo Welt" genau 20 mal untereinander in dem Applet angezeigt wird, und zwar linksbündig mit einem Randabstand von 50 Pixeln.

Wie immer, gilt auch hier die Regel: Je kürzer der Quelltext, desto besser!

Schritt 5 - Linien

Wir wollen jetzt eine **Linie** in das Applet zeichnen, und zwar eine Linie, die von oben links nach rechts unten geht. Der Befehl zum Zeichnen von Linien heißt **drawLine()** und ist eine weitere Methode der Klasse **Graphics**. Ergänzen Sie die **paint()**-Methode Ihres Applets um folgende Zeile:

```
g.drawLine(0,0,500,500);
```



***5.1-4** Eine Linie wird in das Applet gezeichnet*

Man kann auch Figuren mithilfe von Linien zeichnen.

Übung 5.1-2 (2 Punkte)

Schreiben Sie die **paint()**-Methode so um, dass ein rechtwinkliges Dreieck, bestehend aus drei einzelnen Linien, in dem Applet angezeigt wird. "Hallo Welt" soll dann nicht mehr angezeigt werden, und sonst auch nichts, nur das Dreieck.

Schritt 6 - Gitterlinien

Wir wollen jetzt ein Gittermuster in das Applet zeichnen.

Schritt 6.1

Zuerst machen wir uns klar, wie man eine **senkrechte Linie** zeichnet:

```
g.drawLine(100,0,100,500);
```

Bei einer senkrechten Linie stimmen die X-Koordinate des Startpunktes und die des Endpunktes überein, hier im Beispiel haben beide X-Koordinaten den Wert 100. Die Y-Koordinate des Startpunktes liegt ganz oben im Applet, also bei der Position 0, und die Y-Koordinate des Endpunktes liegt entsprechend am unteren Ende, also bei 500.

Schritt 6.2

Wir wollen nun eine zweite senkrechte Linie zeichnen, und zwar bei X = 150:

```
g.drawLine(100,0,100,500);
g.drawLine(150,0,150,500);
```

Hier sehen wir die Befehle für die beiden Linien. Sie stimmen genau überein - nur die X-Werte unterscheiden sich.

Schritt 6.3

Wir wollen jetzt das ganze Applet mit senkrechten Linien ausfüllen. Dazu könnte man jetzt ja einfach neun mal den **drawLine()**-Befehl hinschreiben:

```
g.drawLine( 50,0, 50,500);
g.drawLine(100,0,100,500);
g.drawLine(150,0,150,500);
...
g.drawLine(450,0,450,500);
```

Aber wozu haben wir eigentlich in Folge 4 die for-Schleifen kennengelernt?

```
for (int x = 50; x <= 450; x = x + 50)
   g.drawLine(x,0,x,500);
```

Mit dieser Zeile erreicht man das Gleiche wie mit den neun Zeilen oben. Die Laufvariable wurde aus nahliegenden Gründen **x** genannt, der Startwert wurde auf die X-Koordinate der ersten Linie festgelegt, nämlich 50, und das Inkrement wurde auf den Abstand zwischen den Linien festgelegt, ebenfalls 50.

Auf die gleiche Weise zeichnen wir jetzt die waagerechten Linien:

```
for (int y = 50; y <= 450; y = y + 50)
   g.drawLine(0,y,500,y);
```

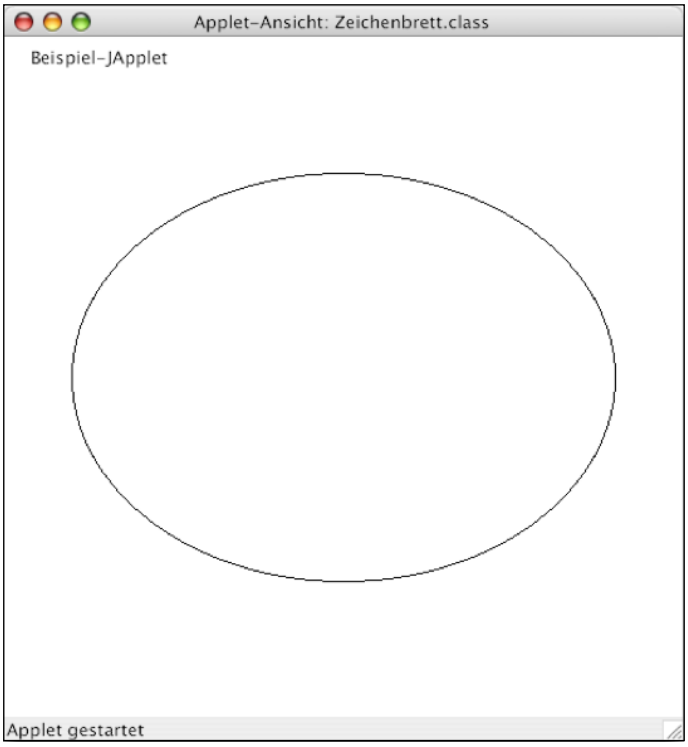
Als Laufvariable wurde jetzt y gewählt, und beim **drawLine()**-Befehl wurden die X-Werte konstant gewählt (0 und 500), während die Y-Werte durch die Laufvariable vorgegeben werden.

Schritt 7 - Kreise

Zum Zeichnen von Kreisen gibt es einen einprägsamen Befehl: **drawOval()**:

```
public void paint(Graphics g)
{
   g.drawString("Beispiel-JApplet", 20, 20);
   g.drawOval(50,100,400,300);
}
```

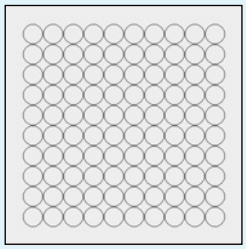
Und so sollte das Applet aussehen, wenn es kompiliert und ausgeführt wurde:



***5.1-5** Ein Oval*

Die Art und Weise, mit der man in Java Kreise und Ovale zeichnet, ist etwas gewöhnungsbedürftig. Normalerweise würde man denken, dass man für einen Kreis die Koordinaten des Mittelpunkts sowie den Radius angeben muss. In Java dagegen gibt man die Koordinaten der linken oberen Ecke des Rechteckes an, das den Kreis / das Oval umschreibt, und außerdem die Breite und die Höhe dieses Rechteckes.

Übung 5.1-3 (3 Punkte)



Schreiben Sie eine möglichst kurze **paint()**-Methode, die ein Applet erzeugt, das ähnlich aussieht wie in der Abbildung: Es sollen zehn Zeilen mit jeweils zehn Kreisen gezeichnet werden.

Es dürfen maximal zwei for-Schleifen oder while-Schleifen dafür benutzt werden.

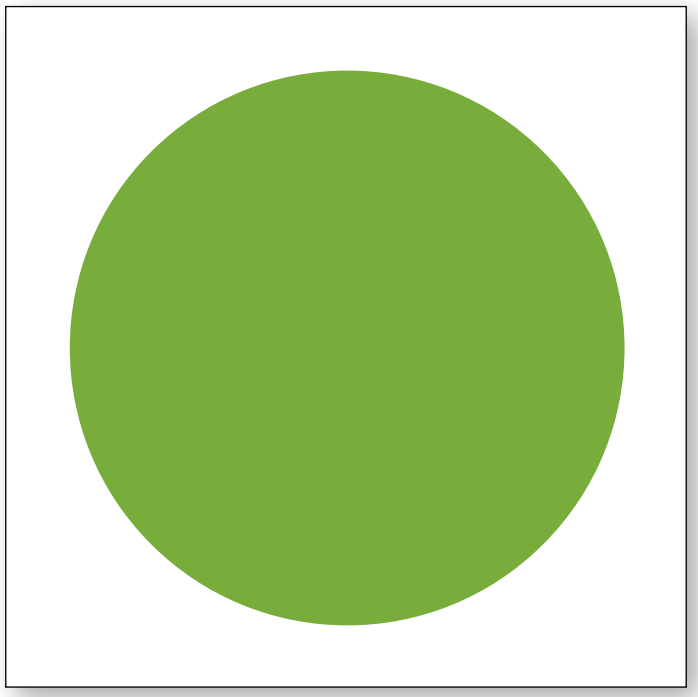
Schritt 8 - Farbige Kreise

In Schritt 7 haben Sie einfache Kreise und Ovale kennen gelernt. Neben dem Befehl **drawOval()** gibt es auch noch den Befehl **fillOval()**, der ausgefüllte Kreise zeichnet. Dieser Befehl wird mit den selben Parametern aufgerufen wie **drawOval()**. Die mit **fillOval()** gezeichneten Ovale haben allerdings *keinen schwarzen Rand*, es wird nur die Innenfläche gezeichnet.

Wir wollen jetzt mal einen grünen Kreis zeichnen:

```
public void paint(Graphics g)
{
   g.setColor(Color.GREEN);
   g.fillOval(50,50,400,400);
}
```

Und so sollte das Applet dann aussehen:



***5.5-6** Ein farbiges Oval ohne Rand*

Die Klasse **Graphics** besitzt eine Methode zum Verändern der aktuellen Zeichenfarbe. Bisher hatten wir **setColor()** noch nicht aufgerufen, daher wurde immer alles in Schwarz gezeichnet. Wollen wir eine andere Zeichenfarbe haben, müssen wir **setColor()** aufrufen und als Parameter die gewünschte Farbe übergeben. Dazu gibt es zwei Möglichkeiten.

Möglichkeit 1: Die Farbkonstanten

Die Klasse **Color** enthält einige Farbkonstanten, die wir direkt in den **setColor()**-Befehl einsetzen können, beispielsweise mit **g.setColor(Color.RED)**:

- Color.**BLACK**
- Color.**DARK_GRAY**
- Color.**GRAY**
- Color.**LIGHT_GRAY**
- Color.**WHITE**
- Color.**MAGENTA**
- Color.**RED**
- Color.**PINK**
- Color.**ORANGE**
- Color.**YELLOW**
- Color.**GREEN**
- Color.**CYAN**
- Color.**BLUE**

Eigentlich müssen die Farbkonstanten (wie bei Konstanten üblich) groß geschrieben werden, so wie oben dargestellt. Man kann die Namen aber auch klein schreiben, also beispielsweise `Color.black` statt `Color.BLACK`. Das Wort "Color" muss allerdings zwingend mit einem großen "C" geschrieben werden, da es sich um den Namen einer Klasse handelt.

Möglichkeit 2: Den Konstruktor der Klasse Color verwenden

Wenn Sie statt dessen den Konstruktor der Klasse **Color** verwenden, können sie viele andere Farbtöne erzeugen, nicht nur die wenigen durch die Farbkonstanten vorgegebenen Farben. Hier zunächst ein Beispiel zur Verwendung des Konstruktors:

```
Color neueFarbe = new Color(134,233,20);
g.setColor(neueFarbe);
g.fillOval(50,50,100,100);
```

Der Konstruktor erwartet drei Parameter, die Sie vielleicht schon aus dem HTML-Kurs kennen. Diese drei Parameter stehen für den Rot-, den Grün- und den Blau-Anteil der Farbe (RGB-System). Jeder Parameter muss einen Wert zwischen 0 und 255 haben. Wenn Sie die Farbe Schwarz erzeugen wollen, schreiben Sie **newColor(0,0,0)**, wenn Sie Weiß haben wollen: **Color(255,255,255)**. Einen schönen Grauton erhalten Sie mit **newColor(127,127,127)**. Wenn Sie ein knalliges Rot haben wollen: **newColor(255,0,0)**, und für ein entsprechendes Grün: **newColor(0,255,0)**. Ich denke, diese Beispiele sollten reichen. Am besten experimentieren Sie mal mit den verschiedenen Farben oder machen die Übung 5.1-4.

Übrigens ist es nicht unbedingt nötig, eine eigene Variable vom Typ **Color** zu verwenden, man kann die neue Farbe auch *direkt* beim Aufrufen des **setColor()**-Befehls erzeugen, wie das folgende Beispiel zeigt:

```
g.setColor(new Color(255,0,127));
g.fillOval(50,50,100,100);
```

Probieren Sie es selbst mal aus!

Schritt 9 - Farbige Kreise mit Rand

Wenn wir den Befehl **drawOval()** verwenden, erhalten wir einen Kreis mit einem Rand, aber ohne Innenfläche. Verwenden wir statt dessen den Befehl **fillOval()**, erhalten wir einen Kreis mit einer Innenfläche, aber ohne Rand.

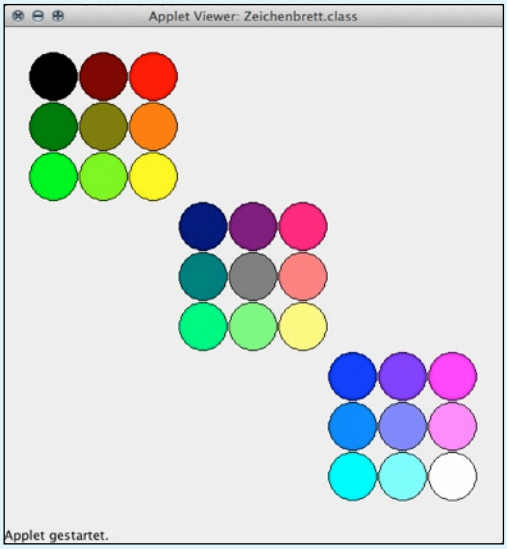
Wir wollen nun einen Kreis mit einem schwarzen Rand und einer farbigen Innenfläche zeichnen. Dazu verwenden wir folgenden Trick: Zunächst zeichnen wir mit **fillOval()** die farbige Innenfläche. Vorher müssen wir die Farbe der Innenfläche natürlich mit **setColor()** bestimmen. Wenn wir den farbigen Kreis gezeichnet haben, zeichnen wir anschließend mit **drawOval()** den schwarzen Rand auf den farbigen Kreis. Vorher müssen wir noch **setColor(Color.BLACK)** aufrufen, damit als Zeichenfarbe Schwarz genommen wird.

Hier der komplette Quelltext:

```
g.setColor(Color.GREEN);
g.fillOval(50,50,400,400);
g.setColor(Color.BLACK);
g.drawOval(50,50,400,400);
```

Übung 5.1-4 (6 Punkte)

Schreiben Sie ein Java-Applet, wie es rechts abgebildet ist, mit möglichst wenig Quelltext.



Die Farben der Kreise haben Werte wie (0,0,0); (0,0,127); (0,0,255); (0,127,0) oder (127,127,255). Es werden also nur die Zahlen 0, 127 und 255 für die Definition der Farbwerte benutzt.

In dem Applet rechts ist der Blau-Anteil im oberen 9er Pack gleich Null, im mittleren 9er Pack gleich 127, und im unteren 9er Pack gleich 255.

Der Rot-Anteil nimmt in den Reihen schrittweise zu, und der Grün-Anteil in den Zeilen.

Einen kurzen Quelltext erreichen Sie, indem Sie drei for-Schleifen verschachteln:

```
for (int i=1; i<=3; i++)
   for (int j=1; j<=3; j++)
      for (int k=1; k<=3; k++)
```

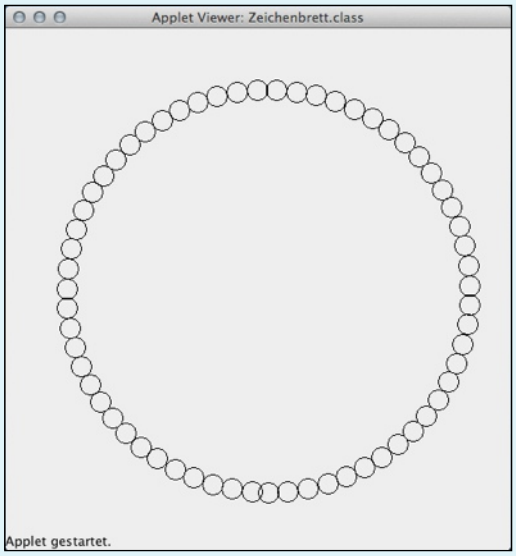
Aus den i, j und k-Werten berechnen sich dann jeweils die x- und y-Koordinaten der Kreise wie auch die R-, G- und B-Werte der Farben.

Expertenübung 5.1-5 (3 Punkte)

In dieser Expertenaufgabe sollen Sie einen Kreis aus Kreisen zeichnen, etwa so wie im Applet rechts gezeigt.

Die drei Punkte erhalten Sie nur dann, wenn Ihr Quelltext nicht wesentlich länger ist als der Quelltext, mit dem ich diese Kreise gezeichnet habe (siehe Abbildung unten).

Hinweis: Sie müssen sich schon etwas mit Sinus und Cosinus sowie der Java-Klasse **Math** auskennen. Deswegen ist dies ja auch eine Expertenübung.




```
import java.awt.*;
import javax.swing.*;

public class Zeichenbrett extends JApplet
{
    public void paint(Graphics g)
    {
        int x,y;

        for (double t = 0; t <= 2*3.14159; t = t+0.1)
        {
           x = 250 + (int) (200 * Math.sin(t));
           y = 250 + (int) (200 * Math.cos(t));
           g.drawOval(x,y,20,20);
        }

    }

}
```

Lösungshinweis:

Informieren Sie sich im Internet über **Lissajous-Figuren**, dann finden Sie wahrscheinlich eine ähnlich kurze Lösung wie in dem unlesbar gemachten Quelltext oben.

Befehlsreferenz

5.2 Wichtige Graphik-Befehle (I3 / I)

drawLine()

Methode der Klasse **Graphics** zum Zeichnen einer Linie.

Syntax:

```
g.drawLine(x1,y1,x2,y2);
```

Parameter (int-Werte):

Alle vier Parameter sind vom Typ **int**. Die Parameter `x1,y1` bestimmen die **Anfangskoordinaten**, die Parameter `x2,y2` die **Endkoordinaten** der Linie.

Linien werden stets in der aktuellen Zeichenfarbe und mit einer Strichstärke von 1 Pixel gezeichnet. Will man eine horizontale (waagerechte) Linie zeichnen, so müssen die beiden y-Werte übereinstimmen. Bei vertikalen (senkrechten) Linien müssen die beiden x-Koordinaten übereinstimmen.

drawRect() und fillRect()

Methode der Klasse **Graphics** zum Zeichnen von Rechtecken.

Syntax:

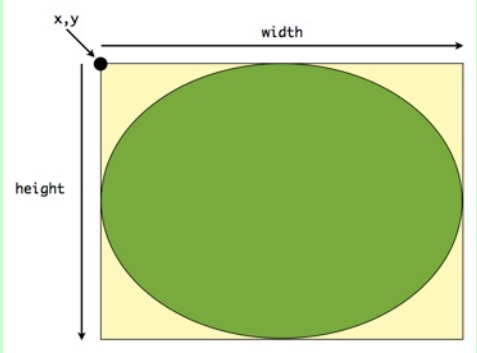
```
g.drawRect(x,y,w,h); g.fillRect(x,y,w,h);
```

Parameter:

Alle vier Parameter sind vom Typ **int**. Die Parameter `x,y` bestimmen die **Koordinaten der linken oberen Ecke**, die Parameter `w,h` legen die **Breite** und die **Höhe** des Rechtecks fest.

Die Methode **drawRect()** zeichnet nur den Rahmen des Rechtecks, innen ist es weiß.
Mit **fillRect()** wird die randlose Innenfläche eines Rechtecks gezeichnet, und zwar in der aktuellen Zeichenfarbe.

drawOval() und fillOval



Methode der Klasse **Graphics** zum Zeichnen eines Ovals.

Syntax:

```
g.drawOval(x,y,w,h);
g.fillOval(x,y,w,h);
```

Parameter:

Die vier int-Parameter definieren nicht das Oval direkt, sondern das *Rechteck*, welches das Oval umschreibt (siehe Zeichnung). Die Parameter `x,y` legen die **Koordinaten der linken oberen Ecke** dieses Rechtecks fest, die Parameter `w,h` die **Breite** und die **Höhe** des Rechtecks.

Die Methoden **drawOval()** und **fillOval()** unterscheiden sich genau so wie **drawRect()** und **fillRect()**. Will man einen *Kreis* zeichnen, so müssen w und h übereinstimmen.

setColor()

Methode der Klasse **Graphics** zum Festlegen der aktuellen Zeichenfarbe. Mit dieser Farbe werden ab jetzt alle Graphik-Objekte gezeichnet, solange, bis **setColor()** erneut aufgerufen wird.

Syntax:

```
g.setColor(color);
```

Parameter:

Der Parameter ist ein Objekt der Klasse **Color**. Es gibt verschiedene Möglichkeiten, wie man ein Color-Objekt erzeugt.

Eine dieser Möglichkeit sehen Sie gleich hier:

Color(r,g,b)

Konstruktor der Klasse **Color**.

Syntax:

```
Color farbe = new Color(r,g,b);
```

Parameter:

Bei den drei Parametern handelt es sich nicht um byte-Werte, also um ganze Zahlen zwischen 0 und 255. Die drei byte-Werte bestimmen den Rot-, Grün- und Blau-Anteil der Farbe.

Unterrichtsvorhaben 2

5.3 Die Klasse Kreis (I1,I2,I3 / A,M,I)

Wir wollen nun eine eigene **Klasse Kreis** konstruieren, mit der man Kreise zeichnen kann, die farbig sind und einen Rand haben, dessen Stärke wir selbst bestimmen können. In dem fakultativen Workshop 7.4 werden wir auf diese Kreis-Klasse zurückkommen.

Schritt 1 - class Kreis

Erzeugen Sie ein neues Projekt mit zwei Klassen. Die Klasse **Kreis** soll dabei eine ganz normale Java-Klasse sein. Die zweite Klasse (zum Beispiel **MeinApplet**) dient zum Testen und Anzeigen der Kreise, darum muss es sich bei dieser Klasse um ein Java-Applet handeln.

Entfernen Sie die überflüssigen Kommentare und Methoden, so dass jede Klasse nur noch den Minimal-Quelltext enthält. Hier der Quelltext der Klasse **Kreis**:

```
public class Kreis
{
    public Kreis()
    {
      // wird noch ergänzt
    }
}
```

Und hier der Quelltext der Klasse **MeinApplet**:

```
import java.awt.*;
import javax.swing.*;

public class MeinApplet extends JApplet
{
   public void init()
   {
      // wird noch ergänzt
   }

   public void paint(Graphics g)
   {
      // wird noch ergänzt
   }
}
```

Die Methode **init()** benötigen wir, um die Objekte der Klasse **Kreis** zu initialisieren. Bisher hatten wir die **init()**-Methode ja immer "weggemacht", jetzt brauchen wir sie aber.

Schritt 2 - Applet mit Graphik-Objekten

Wir wollen die Kreise nun so konstruieren, wie es uns der "gesunde Menschenverstand" vorschreibt. Ein Kreis hat einen **Mittelpunkt** sowie einen **Radius**. Entsprechend soll auch der Konstruktor der Klasse **Kreis** aussehen.

Zunächst ergänzen wir unser Applet so, dass ein Objekt der Klasse **Kreis** erzeugt und eingebunden wird:

```
import java.awt.*;
import javax.swing.*;

public class MeinApplet extends JApplet
{
   Kreis k1;

   public void init()
   {
      k1 = new Kreis(250,250,100);
   }

   public void paint(Graphics g)
   {
      k1.paint(g);
   }
}
```

Kompilieren können Sie das Applet noch nicht, weil die Klasse **Kreis** ja weder einen Konstruktor noch eine **paint()**-Methode besitzt. Aber zumindest haben Sie schon mal eine Vorstellung davon bekommen, wie man ein eigenes Graphik-Objekt in ein Java-Applet einbindet und weshalb die Methode **init()** wichtig für das Applet ist: In der **init()**-Methode werden alle Objekte erzeugt, die in dem Applet verwendet werden. In der **paint()**-Methode dagegen werden dann die **paint()**-Methoden der Graphik-Objekte aufgerufen, die übrigens auch gern andere Namen haben dürfen - wichtig ist nur, dass diese Methoden Zugriff auf die Zeichenfläche des Applets erhalten. Darum ist es unbedingt notwendig, den **paint()**-Methoden der Objekte den Parameter **g** vom Typ **Graphics** zu übergeben.

Top-Down-Verfahren

Wir sind hier übrigens nach dem berühmten **Top-Down-Verfahren** vorgegangen. Zunächst haben wir den groben Ablauf des Programms festgelegt und uns noch nicht um die Einzelheiten der Implementation gekümmert. Das kommt später.

Schritt 3 - Anpassen der Klasse Kreis

Konstruktor und Attribute

Wir wollen die Kreise nun so zeichnen bzw. initialisieren, wie es uns der "gesunde Menschenverstand" vorgibt.

```
public class Kreis
{
    int xPos, yPos, radius;

    public Kreis(int x, int y, int rad)
    {
        xPos = x;
        yPos = y;
        radius = rad;
    }
}
```

Mit dieser Ergänzung hat die Klasse **Kreis** schon mal einen Konstruktor, der die "richtigen" Parameter empfängt, nämlich die Koordinaten des Kreismittelpunktes sowie den Radius. In dem Konstruktor werden diese Parameterwerte dann den entsprechenden Attributen der Klasse übergeben, damit auch andere Methoden Zugriff auf diese Eigenschaften der Kreisobjekte haben.

Das Applet können wir aber immer noch nicht kompilieren, weil die Methode **paint()** der Klasse Kreis noch nicht geschrieben wurde. Das ist jetzt unser nächster Auftrag.

Die paint()-Methode

Wir versuchen jetzt mal Folgendes:

```
public class Kreis
{
    int xPos, yPos, radius;

    public Kreis(int x, int y, int rad)
    {
        xPos = x;
        yPos = y;
        radius = rad;
    }

    public void paint()
    {

    }
}
```

Diesen Quelltext können wir ohne Probleme kompilieren. Was passiert aber, wenn wir jetzt das Applet kompilieren wollen? Es gibt eine Fehlermeldung *"method paint in class Kreis cannot be applied to given types; required: no arguments; found: java.awt.Graphics..."*

Diese Meldung ist eindeutig (wenn man gut Englisch versteht). Die Klasse **Kreis** stellt zwar eine **paint()**-Methode zur Verfügung, was an sich schon mal gut ist. Allerdings erwartet diese **paint()**-

Methode *keinen* Parameter, während das Applet der **paint()**-Methode einen Parameter **g** der Klasse **Graphics** übergeben muss, damit das Kreis-Objekt überhaupt Zugriff auf die Zeichenfläche des Applets bekommt.

"Kein Problem", denken wir jetzt und erweitern die Klasse **Kreis** entsprechend um diesen gewünschten Parameter:

```
    public void paint(Graphics g)
    {

    }
```

Jetzt gibt es eine neue Fehlermeldung beim Kompilieren: Java kann die Klasse **Graphics** nicht finden. Das ist ja auch völlig richtig, denn wir haben unserer **Kreis**-Klasse an keiner Stelle mitgeteilt, *wo* sich die ganzen Graphik-Befehle bzw. die Klasse **Graphics** befindet. Schauen wir uns noch einmal den Quelltext des Applets an:

```
import java.awt.*;
import javax.swing.*;

public class MeinApplet extends JApplet
{
   Kreis k1;
   etc.
```

Die Informationen über die Klasse **Graphics** stecken in der Bibliothek **java.awt**, die mit dem `import`-Befehl in das Applet eingebunden wurde. Das versuchen wir jetzt auch einmal für unsere Klasse **Kreis**:

```
import java.awt.*;

public class Kreis
{
    int xPos, yPos, radius;
    etc.
```

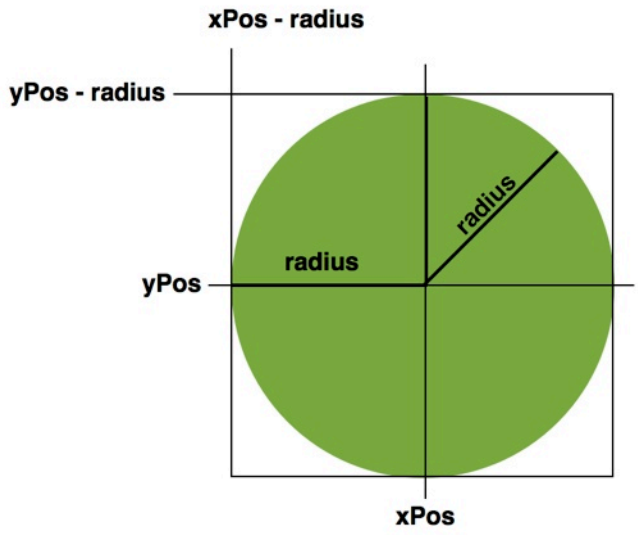
Tatsächlich - jetzt kann der Quelltext problemlos kompiliert werden. Auch das Applet können wir nun kompilieren und ausführen - wir erhalten jetzt eine weiße Fläche, die genau 500 mal 500 Pixel groß ist. Kein Wunder, denn die **paint()**-Methode der Klasse **Kreis** ist ja auch noch leer.

Schritt 4 - die paint()-Methode

Mit der folgenden **paint()**-Methode schaffen wir es endlich, einen großen grünen Kreis in das Applet zu zeichnen:

```
public void paint(Graphics g)
 {
     g.setColor(Color.GREEN);
     g.fillOval(xPos-radius,yPos-radius,radius*2,radius*2);
 }
```

Die folgende Zeichnung erklärt, wie man aus den Attributen **xPos**, **yPos** und **radius** die Parameter für den **fillOval()**-Befehl berechnet:



***5.3-1** Berechnung der Koordinaten*

Die x-Position der linken oberen Ecke des umschreibenden Quadrats ergibt sich aus der x-Position des Kreismittelpunkts, wenn man den Radius subtrahiert. Entsprechendes gilt für die y-Position des Quadrates. Die Breite und die Höhe des Quadrates entsprechen dem doppelten Radius, also dem Durchmesser des Kreises.

Schritt 5 - die Farbe des Kreises bestimmen

Wir wollen jetzt farbige Kreise zeichnen. Dazu ändern wir zunächst wieder unser Applet und passen anschließend die Klasse **Kreis** an:

```
public void init()
{
   k1 = new Kreis(250,250,100,Color.BLUE);
}
```

Das ist der veränderte Quelltext der **init()**-Methode in unserem Applet. Das Applet können wir jetzt nicht mehr kompilieren, wieder ärgert uns eine Fehlermeldung. Das Applet übergibt dem Konstruktor der Klasse **Kreis** *vier* Parameter, der Konstruktor der Klasse **Kreis** erwartet aber nur *drei* Parameter.

Also müssen wir wieder die Klasse **Kreis** verändern:

```
import java.awt.*;

public class Kreis
{
    int xPos, yPos, radius;
    Color farbe;

    public Kreis(int x, int y, int rad, Color col)
    {
        xPos = x;
        yPos = y;
        radius = rad;
        farbe = col;
    }

    public void paint(Graphics g)
    {
        g.setColor(farbe);
        g.fillOval(xPos-radius,yPos-radius,radius*2,radius*2);
    }
}
```

Kleine Übung für zwischendurch:

Übung 5.3-1 (2 Punkte)

Schreiben Sie das Applet so um, dass es zwei Kreise anzeigt; nämlich einen blauen und einen roten. Die beiden Kreise sollen sich nicht überlappen.

Schritt 6 - Kreise mit Rand zeichnen

Ich könnte Ihnen an dieser Stelle jetzt noch genau erklären, wie man die Klasse **Kreis** dazu bringt, farbige Kreise mit einem schwarzen Rand zu zeichnen, aber ich denke, das ist eigentlich schon Stoff für eine weitere Übung.

Im Schritt 9 des ersten Unterrichtsvorhabens haben Sie gelernt, wie man einen farbigen Kreis mit einem Rand zeichnet. Sie sollten also die nächste Übung schaffen können.

Übung 5.3-2 (2 Punkte)

Ergänzen Sie die Klasse **Kreis** so, dass die Kreise einen Rand in einer beliebigen Farbe bekommen. Beim Aufrufen des Konstruktors soll die Innenfarbe und die Randfarbe als Parameter übergeben werden, beispielsweise so:

```
Kreis k1 = new Kreis(200,200,60,Color.GREEN,Color.BLACK);
```

Übung 5.3-3 (2 Punkte)

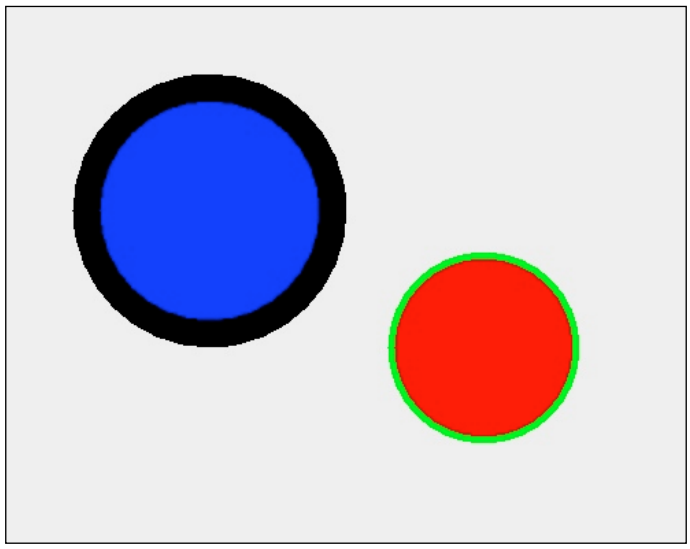
Ergänzen Sie die Klasse **Kreis** so, dass die Kreise einen Rand in einer beliebigen Farbe und in einer beliebigen Randstärke bekommen. Beim Aufrufen des Konstruktors soll die Innenfarbe, die Randfarbe und die Randstärke als Parameter übergeben werden, beispielsweise so:

```
Kreis k1 = new Kreis(200,200,120,Color.GREEN,Color.BLACK,20);
```

Der Gesamtradius des Kreises in dem obigen Beispiel ist weiterhin 120, der grüne Innenkreis hätte hier also einen Radius von 120-20, also von 100 Pixeln.

Schritt 7 - Mehrere Kreise im Applet

Bisher zeichnet unser Applet nur einen einsamen Kreis. Das liegt daran, dass wir nur ein einziges Objekt der Klasse Kreis erzeugt haben. Das Applet lässt sich aber sehr leicht so erweitern, dass auch zwei Kreise in verschiedenen Farben an unterschiedlichen Positionen angezeigt werden:



***5.3-2** Das Applet zeigt zwei Kreise*

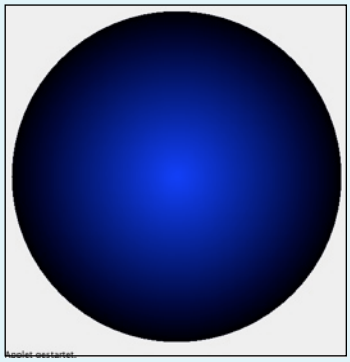
Die obige Abbildung wurde mit folgendem Applet-Quelltext erzeugt:

```
import java.awt.*;
import javax.swing.*;

public class MeinApplet extends JApplet
{
   Kreis k1,k2;

   public void init()
   {
      k1 = new Kreis(150,150,100,Color.BLUE,Color.BLACK,20);
      k2 = new Kreis(350,250, 70,Color.RED,Color.GREEN,5);
   }

   public void paint(Graphics g)
   {
      k1.paint(g);
      k2.paint(g);
   }
}
```

Auf die gleiche Weise kann man das Applet um einen dritten, einen vierten und weitere Kreise ergänzen.

Expertenübung 5.3-4 (3 Punkte)

Sie sollen die Klasse **Kreis** sowie das Applet so ergänzen / verändern, dass das Applet einen großen Kreis mit einem radialen Farbverlauf (siehe rechts) zeigt.



Lösungshinweise

Die Klasse **Kreis** muss auch Kreise darstellen können, deren Randdicke den Wert 0 hat, also Kreise ohne Rand zeichnen können. Das Bild oben besteht nämlich aus ganz vielen Kreisen mit unterschiedlicher Farbe, die übereinander liegen, und die keinen Rand haben.

Damit Sie nicht 100 Kreisobjekte in Ihrem Applet deklarieren und initialisieren müssen, sollten Sie die Klasse **Kreis** um zwei Methoden

```
public void setRadius(int rad)
public void setInnenfarbe(int r, int g, int b)
```

ergänzen. Auf diese Weise kommen Sie im Applet mit einem Objekt der Klasse **Kreis** aus. Sie initialisieren dieses Objekt mit einem Radius von 240 Pixeln in der Mitte des Applets (250,250). Die Innenfarbe dieses Kreisobjekts ist Schwarz, die Randdicke ist Null, daher ist die Randfarbe völlig egal.

In die **paint()**-Methode bauen Sie eine for-Schleife ein, in der der Radius und die Farbe des Kreises verändert wird. Hier der Kopf einer solchen for-Schleife:

```
for (int radius = 240; radius >= 5; radius -= 5)
```

Und nun viel Erfolg!

für Abiturienten

5.4 Beziehungen zwischen Objekten (I1 / A,M,D)

HAT- und KENNT-Beziehungen

Jetzt wird es wieder mal etwas theoretisch. Diejenigen von Ihnen, die ihr Abitur im Fach Informatik machen, müssen wissen, welche **Beziehungen** es zwischen Objekten verschiedener Klassen gibt. Betrachten wir zunächst wieder unser Applet aus dem letzten Unterrichtsvorhaben. Die Klasse **MeinApplet** besitzt zwei Objekte **k1** und **k2** der Klasse **Kreis**:

```
public class MeinApplet extends JApplet
{
   Kreis k1,k2;
   ...
```

Zwischen den beiden Klassen **MeinApplet** und **Kreis** besteht eine **Beziehung**. Nun unterscheidet man in der Fachliteratur mehrere Typen dieser Beziehungen.

Allerdings sind sich die verschiedenen Autoren nicht unbedingt einig, worin der genaue Unterschied zwischen einer HAT-Beziehung und einer KENNT-Beziehung besteht. Mehrere Autoren erklären diesen Unterschied gern mit zwei Beispielen.

Golfplatz-Loch = HAT-Beziehung

Hier gibt es die zwei Klassen **Golfplatz** und **Loch**. **Golfplatz** besitzt viele Objekte der Klasse **Loch**. Nun kann ein Loch allerdings *nicht alleine* existieren, sondern ist immer ein untergeordneter Bestandteil eines ganzen Golfplatzes. Eine solche Beziehung wird von vielen Autoren als **HAT-Beziehung** bezeichnet.

Spielfeld-Ball = KENNT-Beziehung

Kann die "untergeordnete" Klasse jedoch auch alleine existieren, so spricht man von einer **KENNT-Beziehung**. Ein Fußball kann auch ohne Fußballfeld existieren. Darum besteht zwischen den Klassen **Spielfeld** und **Ball** eine KENNT-Beziehung.

Mir persönlich scheint diese Unterscheidung sehr künstlich und etwas weit hergeholt zu sein, ich bevorzuge eine andere Sichtweise, die man ebenfalls in der Fachliteratur findet.

Objekte werden intern erzeugt = HAT-Beziehung

Nach dieser Sichtweise werden bei einer HAT-Beziehung die untergeordneten Objekte direkt in der übergeordneten Klasse erzeugt (beispielsweise in der **init()**-Methode eines Applets). Hier mal ein ganz einfaches Beispiel:

```
public class MeinApplet extends JApplet
{
   Kreis k1, k2;

   public void init()
   {
       k1 = new Kreis(250,250,240); k2 = new Kreis(400,120,80);
   }
}
```

Dieses Beispiel sollte Ihnen bekannt vorkommen. In der Klasse **MeinApplet** werden zwei Objekte der Klasse **Kreis** erzeugt. Die beiden Objekte **k1** und **k2** existieren nur in der Klasse **MeinApplet**, nicht außerhalb. Vielleicht ist das auch gemeint, wenn manche Autoren sagen, bei einer HAT-Beziehung können die untergeordneten Objekte nicht alleine existieren.

Objekte werden extern erzeugt = KENNT-Beziehung

Bei einer KENNT-Beziehung dagegen werden die Objekte der untergeordneten Klasse außerhalb der übergeordneten Klasse erzeugt und dieser dann in Form von Parametern übergeben. Auch hier ein einfaches Beispiel, das ich allerdings erst extra für dieses Skript konstruieren musste.

Stellen Sie sich eine einfache Klasse **Kreis** vor:

```
public class Kreis
{
    int x,y,radius;

    public Kreis(int x, int y, int radius)
    {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```

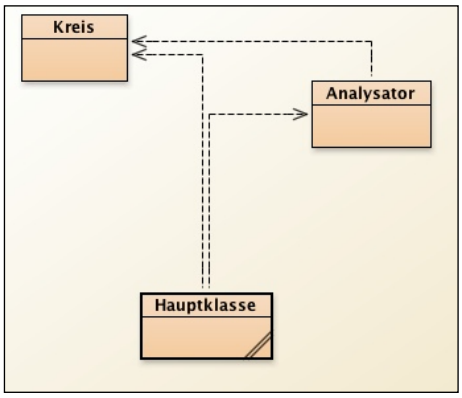
Dann eine Klasse, die ich mal **Analysator** genannt habe, weil mir nichts besseres eingefallen ist:

```
public class Analysator
{
    public int gibDurchmesser(Kreis k)
    {
        return k.radius * 2;
    }
}
```

Und nun schließlich eine dritte Klasse, die ich als **Hauptklasse** bezeichnet habe:

```
public class Hauptklasse
{
    public Hauptklasse()
    {
        Kreis k = new Kreis(100,100,123);
        Analysator a = new Analysator();

        System.out.println("Kreis-Durchmesser = " + a.gibDurchmesser(k));
    }
}
```

Hier sehen Sie das zugehörige BlueJ-Klassendiagramm:



***5.4 - 1** Klassenbeziehungen*

Zwischen der **Hauptklasse** und den Klassen **Kreis** bzw. **Analysator** bestehen HAT-Beziehungen, denn die Objekte **k** und **a** werden in **Hauptklasse** erzeugt.

Aber zwischen dem Objekt **a** und dem Objekt **k** besteht eine KENNT-Beziehung, denn das Objekt **k** der Klasse **Kreis** wird dem Objekt **a** der Klasse **Analysator** als Parameter übergeben. Der **Kreis** wird also nicht im **Analysator** erzeugt, sondern kommt von außen.

HAT-Beziehung

Beziehung zwischen zwei Klassen bzw. Objekten A und B. A besitzt Attribute, die Objekte der Klasse B sind. Diese Objekte werden in der Klasse A erzeugt, existieren also nicht außerhalb der Klasse.

KENNT-Beziehung

Beziehung zwischen zwei Klassen bzw. Objekten A und B. A besitzt Attribute, die Objekte der Klasse B sind. Diese Objekte wurden aber nicht in A erzeugt, sondern von außen nach A übertragen (Parameterübergabe)

Assoziation, Aggregation und Komposition

Neben den Begriffen HAT-Beziehung und KENNT-Beziehung gibt es in der Fachliteratur noch die drei Begriffe Assoziation, Aggregation und Komposition. Dabei entspricht eine **Assoziation** einer KENNT-Beziehung (Objekte werden *außerhalb* der Klasse erzeugt und dann übergeben), eine **Aggregation** einer HAT-Beziehung (Objekte werden *innerhalb* der Klasse erzeugt) und eine **Komposition** einer KENNT-Beziehung nach der alternativen Definition, nach der Objekte nicht außerhalb der Klasse existieren können.

Fazit:

In der Fachliteratur werden die Begriffe HAT-Beziehung und KENNT-Beziehung nicht eindeutig voneinander getrennt. In Wirklichkeit besteht sogar ein heilloses Chaos, was die Begriffe angeht. Daher werde ich jetzt in meinem Skript so gut wie ausschließlich den Begriff HAT-Beziehung verwenden.

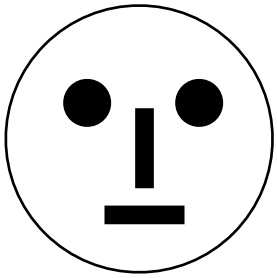
In der graphischen Darstellung von BlueJ wird sowieso nicht zwischen HAT- und KENNT-Beziehungen unterschieden, beide Beziehungen werden durch Pfeile dargestellt, die von der übergeordneten Klasse zur untergeordneten Klasse zeigen.

Unterrichtsvorhaben 3

5.5 Die Klasse Gesicht (I1,I2,I3 / A,M,I)

In dem Unterrichtsvorhaben 2 haben wir eine Graphik-Klasse konstruiert, die einen Kreis repräsentiert. Ein Java-Applet konnte dann mehrere Objekte dieser Klasse **Kreis** besitzen, und durch das Aufrufen der **paint()**-Methoden dieser Kreis-Objekte konnten die Kreise dann auch in das Applet gezeichnet werden.

In diesem Unterrichtsvorhaben wollen wir nun eine etwas komplexere Graphik-Klasse entwickeln, die ihrerseits auf die Klasse **Kreis** zurückgreift. Und zwar soll diese Graphik-Klasse einfache Gesichter zeichnen, daher nennen wir die Klasse einfach mal **Gesicht**.



***5.5 - 1** So könnte ein Gesicht aussehen*

Schritt 1 - Das Top-Down-Prinzip

Bereits im letzten Unterrichtsvorhaben haben wir es angewendet - das **Top-Down-Prinzip** der Software-Entwicklung. Jetzt machen wir es wieder. Wir beginnen mit dem Applet, das die Zeichenfläche für Objekte der Klasse **Gesicht** zur Verfügung stellt. Danach beginnen wir mit der Entwicklung der Klasse **Gesicht**, wobei wir die Klasse zuerst mit leeren Methoden ohne jede Funktion ausstatten, damit das **Applet** schon mal kompiliert werden kann. Erst dann beginnen wir die einzelnen Methoden mit Inhalt zu füllen. Dieses Vorgehen "von oben nach unten" bezeichnet man auch als **Top-Down-Prinzip** der Programmierung bzw. der Software-Entwicklung.

Das Gegenteil wäre das **Bottom-Up-Prinzip**. Hier werden zunächst die untergeordneten Klassen fertig gestellt, und dann wird aus Objekten dieser Klassen eine übergeordnete Klasse konstruiert. Ich persönlich muss zugeben, dass ich meistens nach dem Bottom-Up-Prinzip arbeite, wenn ich komplexere Anwendungen entwickle. Man muss sich nicht von Anfang an Gedanken über das "große Ganze" machen, sondern entwickelt erst mal in Ruhe Klassen, die man *vielleicht* gebrauchen könnte. Erst wenn man mit dieser Arbeit zufrieden ist, geht man eine Stufe höher und baut komplexere Klassen zusammen.

Schritt 2 - Das Applet

Wie bereits gesagt, nach dem Top-Down-Prinzip schrauben wir uns erst mal ein Applet zusammen.

```
import java.awt.*;
import javax.swing.*;

public class TestApplet extends JApplet
{
    Gesicht ges;

    public void init()
    {
       ges = new Gesicht();
    }

    public void paint(Graphics g)
    {
        ges.paint(g);
    }
}
```

Schritt 3 - Die leere Klasse Gesicht

Nun erzeugen wir eine leere Klasse **Gesicht**, damit das obige Applet kompiliert werden kann.

```
import java.awt.*;

public class Gesicht
{
    public Gesicht()
    {
    }

    public void paint(Graphics g)
    {
    }
}
```

Die **paint()**-Methode ist wichtig, weil sie im Applet aufgerufen wird, und der `import`-Befehl ist auch wichtig, weil sonst die Klasse **Graphics** nicht zur Verfügung steht.

Für diese Art der Programmierung gibt es übrigens auch bestimmt einen Fachbegriff, der mir aber gerade entfallen ist. Man sorgt stets dafür, dass das gerade bearbeitete Projekt kompilierbar ist. Sobald man kleine Verbesserungen eingebaut hat, sieht man wieder zu, dass das Projekt kompiliert werden kann, auch wenn noch gar nichts oder zumindest nicht viel angezeigt werden kann.

Schritt 4 - Ein erster Funktionstest

In diesem Schritt wollen wir testen, ob die Klasse **Gesicht** auch tatsächlich in das Applet eingebunden wird und ob die Übergabe der Zeichenfläche an die Klasse **Gesicht** funktioniert. Dazu statten wir die **paint()**-Methode des Gesichts mit einem einfachen Graphik-Befehl aus, den wir später wieder löschen. Er dient ja nur zur Kontrolle, ob die Zusammenarbeit zwischen den Klassen funktioniert.

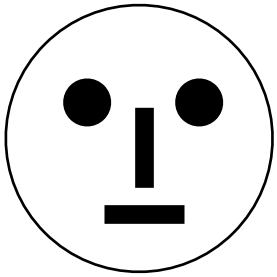
```
import java.awt.*;

public class Gesicht
{
    public Gesicht()
    {
    }

    public void paint(Graphics g)
    {
        g.drawOval(200,200,100,100);
    }
}
```

Jetzt wird in dem Applet tatsächlich ein schwarzer Kreis angezeigt, demnach scheint bisher alles zu funktionieren.

Man könnte jetzt natürlich die **paint()**-Methode von **Gesicht** derart mit Graphik-Befehlen auffüllen, dass tatsächlich ein Gesicht gezeichnet wird. Das ist aber nicht der Sinn dieses Unterrichtsvorhabens. Ich möchte Ihnen in diesem Unterrichtsvorhaben zeigen, wie man mehrere unterschiedliche Klassen dazu "überredet", zusammen zu arbeiten, so das ein "größeres Ganzes" entsteht.

Schritt 5 - Das Gesicht hat Kreise

Betrachten wir das Gesicht noch einmal:



***5.5 - 2** Ein sehr einfaches Gesicht mit drei Kreisen*

Es besteht aus einem großen Kreis und zwei kleinen, gefüllten Kreisen. Außerdem sieht man noch zwei Rechtecke, aber darauf wollen wir erst später eingehen.

Wozu haben wir eigentlich im letzten Unterrichtsvorhaben die Klasse **Kreis** entwickelt? Damit man ohne viel Aufwand Kreise mit einem farbigen Rand beliebiger Dicke und einer anderen Innenfarbe zeichnen kann. Wir wollen unsere Klasse **Gesicht** jetzt mit Objekten der Klasse **Kreis** ausstatten.

```
import java.awt.*;

public class Gesicht
{
    Kreis kopf, augeL, augeR;

    public Gesicht()
    {
    }

    public void paint(Graphics g)
    {
        g.drawOval(200,200,100,100);
    }
}
```

Unsere Klasse **Gesicht** hat jetzt drei Objekte der Klasse **Kreis** - es besteht also eine HAT-Beziehung zwischen den Klassen **Gesicht** und **Kreis.**

Wenn Sie eine Klausur in Informatik schreiben oder gar in diesem Fach ihr Abitur machen wollen, sollten Sie sich unbedingt den Abschnitt 5.4 "Beziehungen zwischen Objekten" ansehen, den ich speziell für die Klausurkandidaten und vor allem für die Abiturienten geschrieben habe.

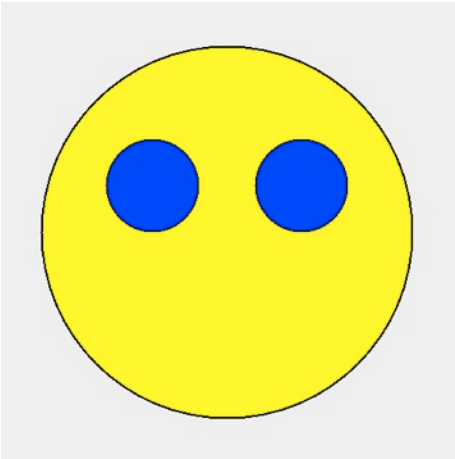
Zwischen den Klassen **MeinApplet** und **Gesicht** existiert ebenfalls eine echte HAT-Beziehung, denn das Objekt **ges** wird in der Klasse **MeinApplet** erzeugt.

Schritt 6 - Initialisierung der drei Kreise

Wir wollen nun die drei bereits deklarierten **Kreis**-Objekte initialisieren. Dafür ist der Konstruktor der Klasse **Gesicht** zuständig.

```
import java.awt.*;

public class Gesicht
{
    Kreis kopf, augeL, augeR;

    public Gesicht()
    {
        kopf  = new Kreis(250,250,200,Color.yellow,Color.black,2);
        augeL = new Kreis(170,200, 50,Color.blue,Color.black,1);
        augeR = new Kreis(330,200, 50,Color.blue,Color.black,1);
    }

    public void paint(Graphics g)
    {
        kopf.paint(g);
        augeL.paint(g);
        augeR.paint(g);
    }
}
```

Die **drawOval()**-Methode, die wir ja nur provisorisch eingesetzt hatten, ersetzen wir nun durch die **paint()**-Methoden der **Kreis**-Objekte.



***5.5 - 3** Das erste Gesicht*

Schritt 7 - Die beiden Rechtecke

Eine Klasse **Rechteck** gibt es noch nicht; auch die Klasse **Kreis** mussten wir uns ja selbst schreiben. Aber ich will Ihnen in diesem Unterrichtsvorhaben nicht alles selbst vormachen, daher erst mal eine kleine Übung:

Übung 5.5-1 (4 Punkte)

Schreiben Sie eine Klasse **Rechteck** analog zur Klasse **Kreis**.

Es sollen also nicht nur die Koordinaten des Rechtecks sowie die Breite und Höhe bestimmt werden können, sondern auch die Randfarbe, die Randstärke sowie die Innenfarbe des Rechtecks.

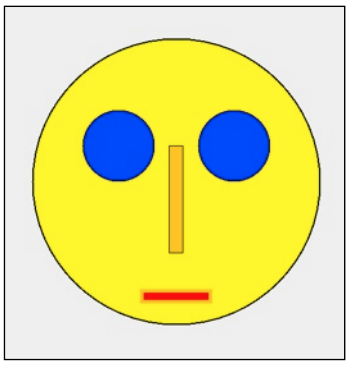
Schritt 8 - Das ganze Gesicht

Auch diesen Schritt überlasse ich Ihnen:

Übung 5.5-2 (2 Punkte)

Ergänzen Sie die Klasse **Gesicht** um zwei Objekte **mund** und **nase** der neuen Klasse **Rechteck**, initialisieren Sie die Objekte und rufen Sie die **paint()**-Methoden der beiden Objekte auf, so dass im Applet schließlich ein komplettes Strichmännchen-Gesicht gezeichnet wird.

Hier sehen Sie das Ergebnis meiner eigenen Bemühungen:



***5.5 - 4** Ein einfaches Gesicht mit Nase und Mund*

Übung 5.5-3 (3 Punkte)

Machen Sie das Gesicht schöner. Die Augen könnten zum Beispiel einen weißen Augapfel und eine bunte Iris haben. Was ist mit Ohren? Ihnen fällt bestimmt noch etwas Schönes ein!

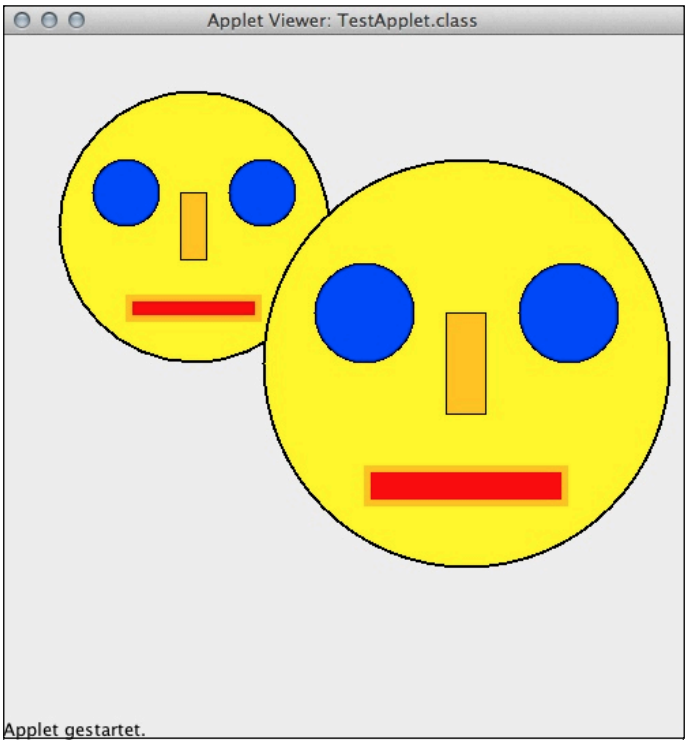
Bitte keine Veränderung des Applets - alle Ergänzungen müssen in den Klassen **Kreis**, **Rechteck** oder **Gesicht** vorgenommen werden. Weitere notwendige Graphik-Klassen dürfen Sie gern ergänzen, wenn Sie wollen. **Extra-Punkte für besonders gute Ideen!**

Expertenübung 5.5-4 (3 Punkte)

Machen Sie das Gesicht flexibler. Bisher hat es ja eine festgelegte Größe; für viele Anwendungen wäre das Gesicht zu groß. Der Benutzer soll selbst entscheiden können, welchen Radius der Kopf (also der äußere Kreis) hat, und entsprechend sollen die anderen Teile des Gesichts dann automatisch kleiner dargestellt werden, und natürlich weiterhin an den passenden Positionen. Das Gesicht soll also, unabhängig von der Größe, immer gleich aussehen, zumindest ungefähr.

Konkret: Der Konstruktor der Klasse **Gesicht** soll als Parameter die Position (x,y) sowie den Radius des Kopfkreises erhalten - den Rest soll die Klasse **Gesicht** von selbst erledigen.

Hier ein Screenshot von einem Applet, das zwei Objekte der Klasse **Gesicht** mit unterschiedlichen Größen eingebunden hat:



***5.5 - 5** Zwei Gesichter in einem Applet*

Unterrichtsvorhaben 4

5.6 Ein Funktionsplotter (I1,I2,I3 / A,M,I)

In diesem Unterrichtsvorhaben wollen wir uns ein kleines Applet bauen, mit dem wir einfache mathematische Funktionen der Art y = f(x) betrachten können. Auf eine Eingabe von Werten durch den Benutzer verzichten wir erst noch mal. Das können wir später mal machen, wenn wir uns besser mit Textfeldern und Buttons auskennen, vielleicht am Ende der Folge 6.

Schritt 1 - Grundidee

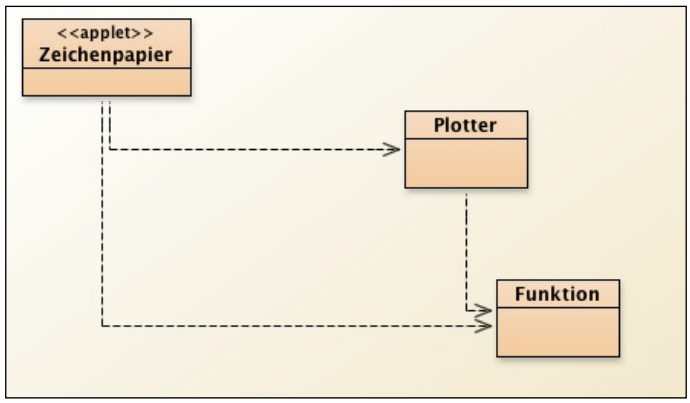
Für die Definition der Funktion erzeugen wir eine eigene Klasse **Funktion**, in der nichts anderes gemacht wird, als die Funktion festzulegen. Der Quelltext dieser Klasse ist sehr einfach:

```
public class Funktion
{
    public double y(double x)
    {
       return x*x;
    }
}
```

Die Methode **y()** legt hier eine quadratische Funktion $y = x^2$ fest.

Dann brauchen wir eine Klasse **Plotter**, welche den Funktionsgraphen zeichnet. Der Einfachheit halber legen wir den Definitionsbereich auf den Bereich von -10 bis +10 und die Schrittweite auf 0.1 fest. Wir sollten dann nur solche Funktionen zeichnen lassen, deren Werte nicht den Rahmen des Applets sprengen.

Schließlich brauchen wir wieder ein **Applet**, damit der Plotter tätig werden kann. Das folgende Klassendiagramm zeigt noch einmal die Zusammenhänge:



***5.6 - 1** Der Funktionsplotter in seiner ersten Version*

Schritt 2 - Das Applet

```
import java.awt.*;
import javax.swing.*;

public class Zeichenpapier extends JApplet
{
    Plotter  plot;

    public void init()
    {
        plot = new Plotter(new Funktion());
    }

    public void paint(Graphics g)
    {
       plot.paint(g);
    }
}
```

Interessant ist hier die Art und Weise, wie die Funktion dem Plotter übergeben wird. Ähnlich wie beim Erzeugen einer neuen Farbe mit

```
g.setColor(new Color(100,100,20));
```

wird hier ein Objekt der Klasse **Funktion** direkt als Parameter für den Konstruktor der Klasse **Plotter** erzeugt. Alternativ hätte man natürlich auch erst ein eigenes Funktions-Objekt erzeugen und dieses dann an den Plotter-Konstruktor übergeben können:

```
func = new Funktion();
plot = new Plotter(func);
```

Schritt 3 - Der Plotter

```
import java.awt.*;

public class Plotter
{
    Funktion func;

    public Plotter(Funktion f)
    {
       func = f;
    }

    public void paint(Graphics g)
    {
    }
}
```

Hier wird auch noch nicht viel gemacht - aber Geduld - wir sind ja erst in der Version 1 unsers Plotters. Gemäß der Philosophie der Top-Down-Programmierung machen wir erst mal den groben Ablauf des Programms klar, bevor wir uns an die Einzelheiten der Implementierung setzen. Und wir achten immer darauf, dass das Programm bzw. das Projekt kompilierbar ist. Das ist ganz wichtig.

Schritt 4 - Das Koordinatensystem

Hier stellt sich die wichtige Frage: sollen wir für die Darstellung der Koordinaten eine eigene neue Klasse entwerfen, oder soll die Klasse **Plotter** diese Funktion übernehmen?

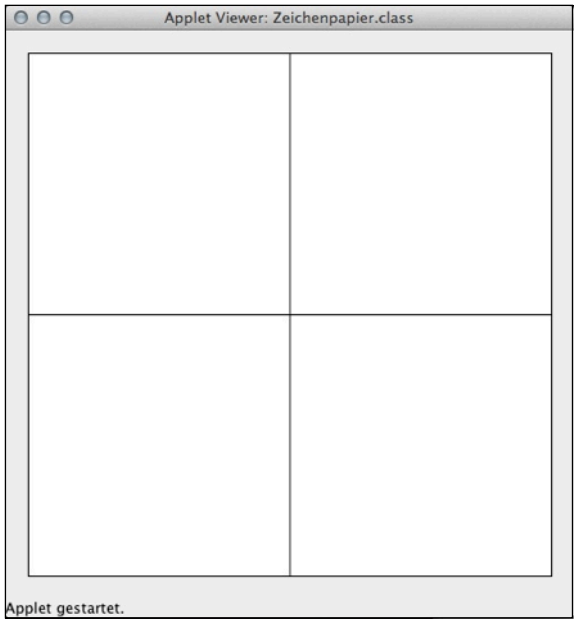
Vielleicht ist es ganz sinnvoll, für die Klasse **Plotter** eine Methode **zeichneKoordinatensystem()** zu schreiben, welche diese Aufgabe übernimmt, und die Methode **paint()** dann nur zum Zeichnen des Funktionsgraphen zu verwenden.

```
public void zeichneKoordinatensystem(Graphics g)
{
   g.drawLine(0,250,500,250); // x-Achse
   g.drawLine(250,0,250,500); // y-Achse
}
```

Mit dieser Methode der Klasse Plotter werden nur die beiden Achsen des Koordinatensystems mitten in das 500 x 500 Pixel große Applet gezeichnet. Vielleicht sollte man noch einen Rahmen um das Koordinatensystem zeichnen, das sieht dann besser aus, falls jemand das Applet mit der Maus anfasst und vergrößert:

```
public void zeichneKoordinatensystem(Graphics g)
{
   g.setColor(Color.WHITE);
   g.fillRect(20,20,460,460);
   g.setColor(Color.BLACK);
   g.drawRect(20,20,460,460);
   g.drawLine(20,250,480,250); // x-Achse
   g.drawLine(250,20,250,480); // y-Achse
}
```

Mit diesen Verbesserungen sieht das Applet schon recht gut aus:



***5.6 - 2** Das Applet mit den Verbesserungen*

Schritt 5 - Maßstäbe, Transformationen und überhaupt...

Jetzt müssen wir ganz wichtige Überlegungen durchführen. Wenn wir die Funktion $y = x^2$ im Definitionsbereich von -10 bis +10 zeichnen wollen, sind ja Funktionswerte zwischen 0 und 100 zu erwarten. Diese Funktionswerte müssen wir nun in Pixelkoordinaten transformieren. Der Funktionswert 0 muss beispielsweise exakt auf der waagerechten Linie in einer Höhe von 250 Pixeln im Applet gezeichnet werden, denn genau dort verläuft die waagerechte Achse des Koordinatensystems. Wir merken uns also:

Der Funktionswert y = 0 entspricht dem Pixelwert y = 250.

Weiter in den Überlegungen. Der Funktionswert 100, der maximal mögliche Funktionswert also, sollte noch in dem weißen Rechteck dargestellt werden können. Der obere Rand des weißen Rechtecks beginnt bei dem Pixelwert y = 20, ist also 250 - 20 = 230 Pixel von der waagerechten Achse entfernt. Wenn wir den Funktionswert y = 100 bei dem Pixelwert y = 50 zeichnen, vereinfacht sich die Sache etwas, und wir haben noch ein bisschen Freiraum nach oben.

Der Funktionswert y = 100 entspricht dem Pixelwert y = 50.

Welchen Pixelwert hätte dann ein Funktionswert wir y = 64?

Nun, die Rechnung ist eigentlich nicht schwer. 100 Einheiten der "richtigen" Funktion entsprechen genau 200 Pixeln. Also müssen wir für eine Einheit der "richtigen" Funktion exakt 2 Pixel berücksichtigen. Der Funktionswerte y = 64 muss daher 128 Pixel oberhalb der waagerechten Achse gezeichnet werden, also bei 250 - 64 = 186 Pixel.

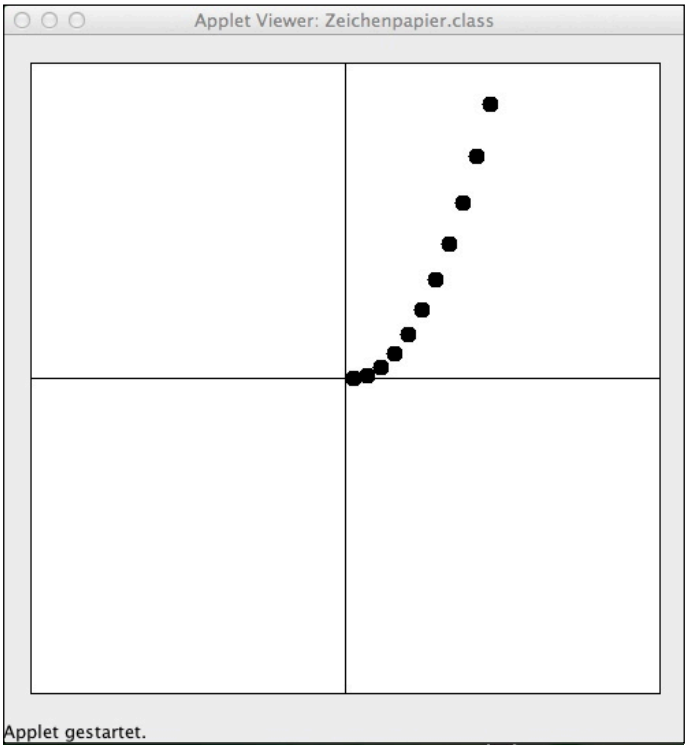
Am besten schreiben wir uns dafür wieder mal eine Java-Methode für die Klasse **Plotter**:

```
private int getYPixel(double y)
{
   return (int) (250 - (y*2));
}
```

Diese Methode testen wir nun mal in der **paint()**-Methode der Klasse **Plotter**:

```
public void paint(Graphics g)
{
    g.fillOval(250,getYPixel(0)-6,12,12);
    g.fillOval(260,getYPixel(1)-6,12,12);
    g.fillOval(270,getYPixel(4)-6,12,12);
    g.fillOval(280,getYPixel(9)-6,12,12);
    g.fillOval(290,getYPixel(16)-6,12,12);
    g.fillOval(300,getYPixel(25)-6,12,12);
    g.fillOval(310,getYPixel(36)-6,12,12);
    g.fillOval(320,getYPixel(49)-6,12,12);
    g.fillOval(330,getYPixel(64)-6,12,12);
    g.fillOval(340,getYPixel(81)-6,12,12);
    g.fillOval(350,getYPixel(100)-6,12,12);
}
```

Ich habe hier einfach mal die Quadratzahlen von 0^2 bis 10^2 als Parameter für die **getYPixel()**-Methode verwendet und das Ergebnis von **getYPixel()** dann als y-Wert für die **fillOval()**-Methode der Klasse **Graphics** genommen, so dass ein 12 Pixel durchmessender Kreis an die betreffende Stelle gezeichnet wird. Das Ergebnis kann sich schon sehen lassen:



***5.6 - 3** Das Applet mit den Test-Kreisen*

Schritt 6 - Das Gleiche mit den X-Werten

Die X-Werte sind hier noch willkürlich gewählt. Auch hier müssen wir jetzt überlegen und die "echten" X-Werte in Pixelwerte umwandeln.

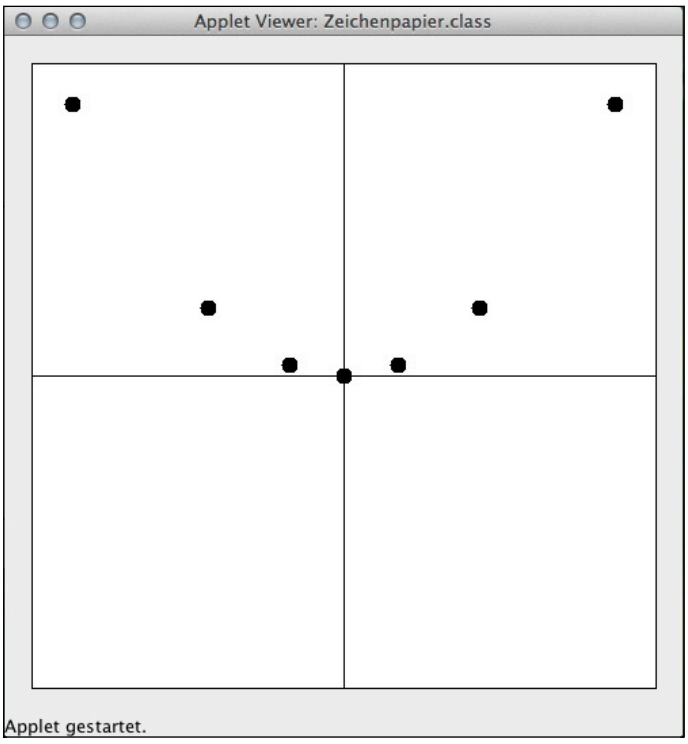
Der "echte" Wert -10 soll bei einem Pixelwert von x = 50 gezeichnet werden. Dann ist nach links hin noch etwas "Reserve", und das Rechnen ist wieder vereinfacht, weil jetzt 10 "echte" Einheiten 200 Pixeln entsprechen, also rechnen wir für eine "echte" Einheit 20 Pixel. Wir schreiben wieder eine Methode dafür:

```
private int getXPixel(double x)
{
   return (int) (250 + x*20);
}
```

Das Typecasting mit `(int)` ist sinnvoll, weil die Pixelwerte immer ganzzahlig sein müssen, der übergebene Parameter aber eine reelle Zahl ist. Nun wollen wir beide Methode wieder testen:

```
public void paint(Graphics g)
 {
     g.fillOval(getXPixel(-10)-6,getYPixel(100)-6,12,12);
     g.fillOval(getXPixel(-5)-6, getYPixel(25)-6,12,12);
     g.fillOval(getXPixel(-2)-6, getYPixel(4)-6,12,12);
     g.fillOval(getXPixel(0)-6,  getYPixel(0)-6,12,12);
     g.fillOval(getXPixel(2)-6,  getYPixel(4)-6,12,12);
     g.fillOval(getXPixel(5)-6,  getYPixel(25)-6,12,12);
     g.fillOval(getXPixel(10)-6, getYPixel(100)-6,12,12);
 }
```

Wir erhalten durchaus eine schöne Graphik:



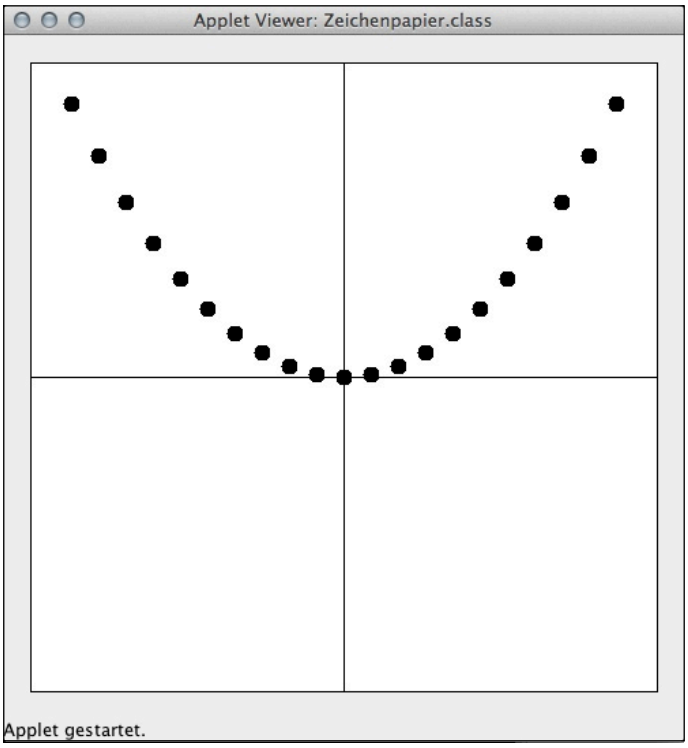
***5.6 - 4** Das Ergebnis der paint()-Methode*

Schritt 7 - Endlich plotten...

Nun wird es Zeit, dass der Funktionsplotter in Aktion tritt.

```
    public void paint(Graphics g)
    {
        for (int x = -10; x <= 10; x++)
           g.fillOval(getXPixel(x)-6,getYPixel(func.y(x))-6,12,12);
    }
```

Mit dieser einfachen **paint()**-Methode in der Klasse **Plotter** erzeugen wir schon mal einen ganz netten Funktionsgraphen:



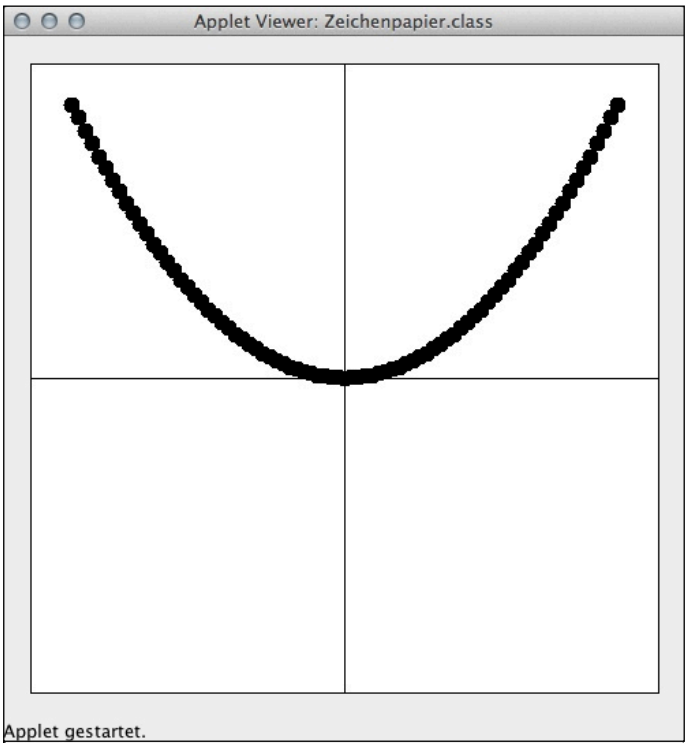
***5.6 - 5** Ein erster Plott*

Schritt 8 - es geht noch besser

Übung 5.6-1 (1 Punkt)

Schreiben Sie die **paint()**-Methode des Plotters so um, dass der X-Wert um jeweils 0,25 erhöht wird.

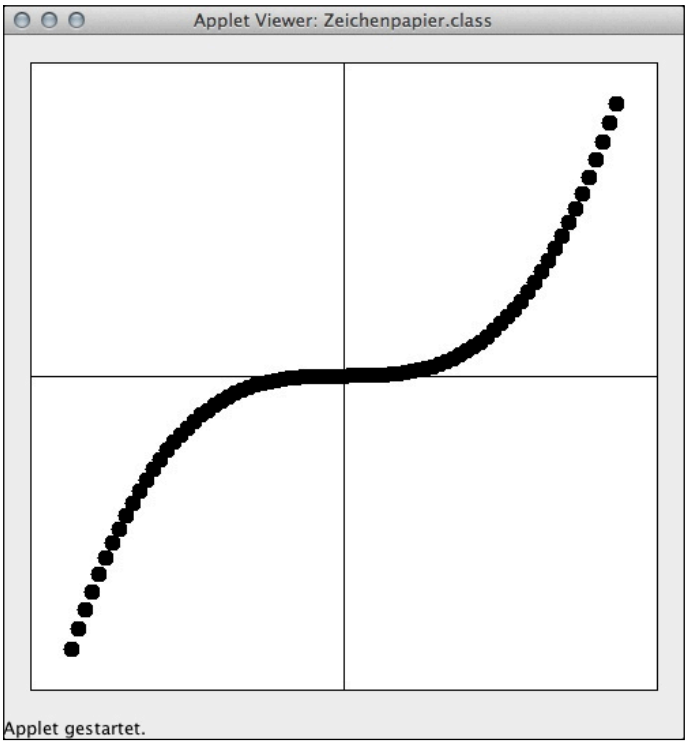
Das Ergebnis kann sich dann schon sehen lassen:



***5.6 - 6** Das Ergebnis der Übung 5.6-1*

Übung 5.6-2 (1 Punkt)

Schreiben Sie die Funktion y in der Klasse **Funktion** so um, dass $y = x^3/10$ geplottet wird. Es soll also eine kubische Funktion gezeichnet werden. Das Dividieren durch 10 ist notwendig, damit der maximale y-Wert nicht über 100 bzw. unter -100 liegt.



***5.6 - 7** Das Ergebnis der Übung 5.6-2*

Schritt 9 - weitere Verfeinerungen

Bei einer Funktion wie y = x^3 bekommen wir sehr große X-Werte in unserem Definitionsbereich, nämlich zwischen -1000 und +1000. Bei anderen Funktionen kann der Wertebereich sogar noch größer sein. Mit der Methode

```
private int getYPixel(double y)
{
   return (int) (250 - (y*2));
}
```

kommen wir dann nicht weit. Den Faktor im Ausdruck 250 - y*2 müssen wir davon abhängig machen, wie groß der maximale Y-Wert der Funktion im Definitionsbereich ist. Und dazu schreiben wir uns - wie immer - eine neue Methode für die Klasse **Plotter**.

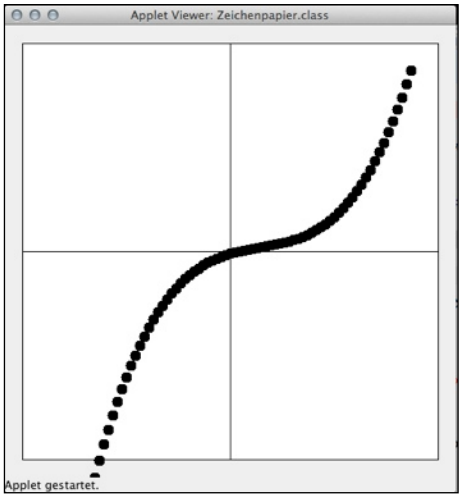
```
private double getMaxY()
{
    double y, maxY = 0;

    for (double x = -10; x <= 10; x += 0.25)
    {
       y = func.y(x);
       if (y > maxY) maxY = y;
    }
    return maxY;
}
```

Diese Methode durchläuft den ganzen Definitionsbereich der Funktion in 0,25er Schritten und vergleicht den Y-Wert mit dem bisherigen maximalen Y-Wert, der in der lokalen Variablen **`maxY`** gespeichert ist. Am Anfang setzen wir diese Variable auf den Wert 0, und jedes Mal, wenn die for-Schleife einen größeren Wert findet, wird **`maxY`** aktualisiert und auf diesen größeren Wert gesetzt.

Am Ende der for-Schleife ist der maximale Y-Wert in **`maxY`** gespeichert und kann mit dem **`return`**-Befehl zurück gegeben werden.

Schauen wir uns nun an, wie die Funktion y = x^3 - 5x^2 +20x -10 geplottet wird, die in der Methode **y()** der Klasse **Funktion** verwaltet wird:

```
public double y(double x)
{
   return x*x*x - 5*x*x + 20*x - 10;
}
```



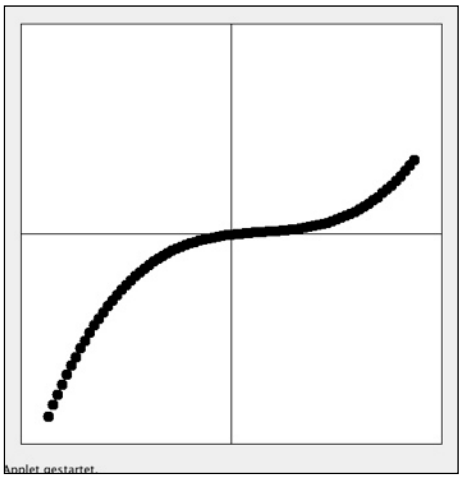
***5.6 - 8** Der Graph der Funktion* $y = x^3 - 5x^2 + 20x - 10$

Die obere Hälfte des Graphen ist in Ordnung, denn die Funktion bleibt innerhalb des Wertbereichs. In der unteren Hälfte verlässt die Funktion allerdings den zuvor festgelegten Bereich, weil die Werte zu negativ werden.

Übung 5.6-3 (2 Punkte)

Verändern bzw. ergänzen Sie die Klasse Plotter so, dass der Funktionsgraph weder nach oben noch nach unten "überläuft". Es reicht also nicht aus, nur den maximalen Funktionswert innerhalb des Definitionsbereichs zu bestimmen.

So sollte der Funktionsgraph dann aussehen:



***5.6 - 9** Der Graph der Funktion* $y = x^3 - 5x^2 + 20x - 10$ *nach der Übung 5.6-3*

Übung 5.6-4 (3 Punkte)

Der Konstruktor der Klasse **Plotter** soll dazu einen zusätzlichen Parameter vom Typ `double` erhalten, der die "Breite" des Definitionsbereichs festlegt:

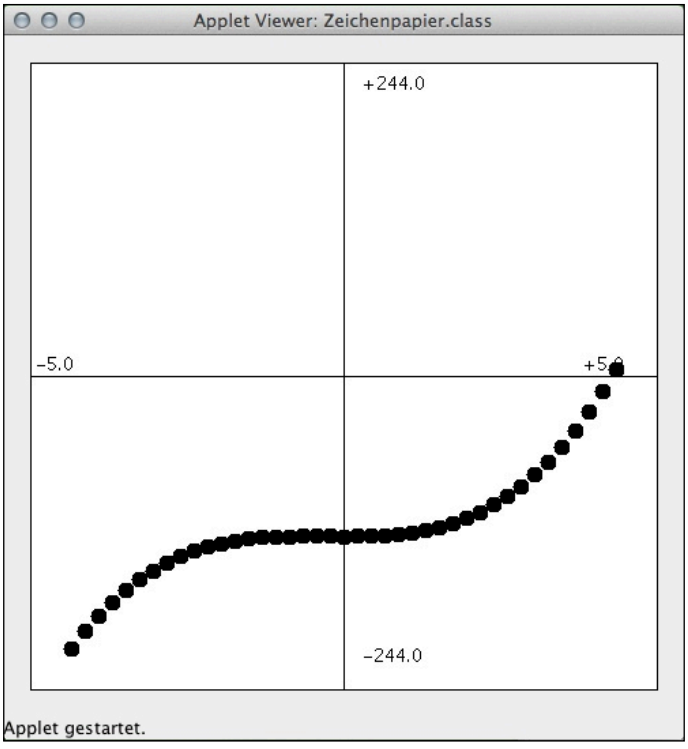
```
public Plotter(Funktion f, double def)
```

Wird als Parameter beispielsweise der Wert 30 übergeben, so erstreckt sich der Definitionsbereich von -30 bis +30. Wird als Wert 4.3 eingegeben, so geht der Definitionsbereich von -4.3 bis +4.3 und so weiter.

Passen Sie die anderen Methoden der Klasse **Plotter** entsprechend an!

Übung 5.6-5 (3 Punkte)

Lassen Sie von der **paint()**-Methode der Klasse **Plotter** die Werte für den Definitionsbereich und für den Wertebereich an die Achsen des Koordinatensystems schreiben, so wie in der Abbildung unten für die Funktion $y = x^3 + x^2 - 144$ im Definitionsbereich -5 ... +5 dargestellt.



5.6 - 10 *Die Funktion* $y = x^3 + x^2 - 144$ *im Definitionsbereich -5 ... +5*

Folge 6 - Ein kleiner Roboter

Wir wollen jetzt ein Java-Applet erstellen, das einen kleinen Roboter zeigt, den wir mithilfe von mehreren Buttons steuern können. Als erstes werden wir eine Klasse **Roboter** erzeugen, dann werden wir uns überlegen, wie man Buttons in einem Applet darstellt und abfragt, und schließlich werden wir die Robotersteuerung implementieren.

Unterrichtsvorhaben 1

6.1 Einen Roboter zeichnen (I1,I2,I3,I5 / A,M,I)

Einstiegsübung (6 Punkte)

Recherchieren Sie, in welchen Gebieten heute Roboter eingesetzt werden, welche Arten von Robotern es gibt und welche Vor- und Nachteile der Einsatz von Robotern hat. Stellen Sie Ihre Ergebnisse im Kurzvortrag vor dem Kurs vor.

Schritt 1 - Die Klasse Roboter erzeugen

Starten Sie BlueJ und erzeugen Sie ein neues leeres Projekt. Anschließend erstellen Sie eine neue Klasse **Roboter**:

```
import java.awt.*;

public class Roboter
{
   int xPos, yPos;

   public Roboter(int x, int y)
   {
      xPos = x;
      yPos = y;
   }

   public void zeigen(Graphics g)
   {
      g.drawOval(xPos-20,yPos-20,40,40);
   }
}
```

Es handelt sich um die erste Version der Klasse **Roboter**. Als Attribute sind lediglich die X-Position und die Y-Position definiert, und neben dem Konstruktor gibt es nur eine Methode zum Anzeigen des Roboters in der Zeichenfläche **g** des Applets. Der Roboter wird in dieser ersten Version durch einen einfachen Kreis mit dem Radius 20 repräsentiert.

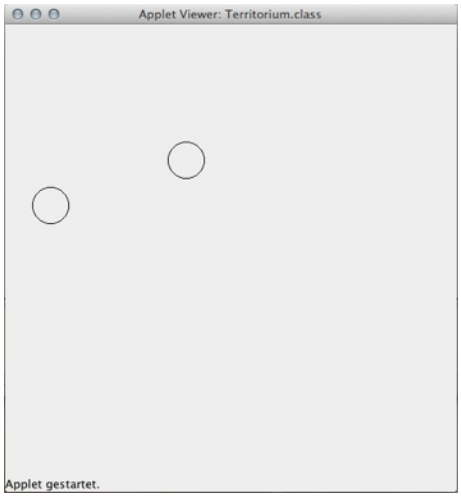
Alternativ hätte man hier ein Objekt der Klasse **Gesicht** aus dem letzten Unterrichtsvorhaben nehmen können, aber aus verschiedenen Gründen, die sich erst später erschließen, wenn wir den Roboter bewegen wollen, wäre dieses Vorgehen doch reichlich umständlich gewesen.

Schritt 2 - Ein Test-Applet

Als nächstes erstellen Sie ein Applet, damit Sie die neue Klasse testen können. Die Objekte der Klasse **Roboter** sollen sich in diesem Applet bewegen können, daher geben wir dem Test-Applet den Klassennamen **Territorium**.

```
import java.awt.*;
import javax.swing.*;

public class Territorium extends JApplet
{
    Roboter robbi1, robbi2;

    public void init()
    {
      robbi1 = new Roboter(50,200);
      robbi2 = new Roboter(200,150);
    }

    public void paint(Graphics g)
    {
      robbi1.zeigen(g);
      robbi2.zeigen(g);
    }
}
```

Das alles sollte Ihnen recht bekannt vorkommen, wenn Sie den Unterrichtsvorhaben zur Klasse Kreis gründlich durchgearbeitet haben. Eine Kurzbeschreibung des Applets ist daher nicht notwendig.



***6.1 - 1** Zwei Objekte der Klasse Roboter*

Schritt 3 - Eine Schönheitskur für den Roboter

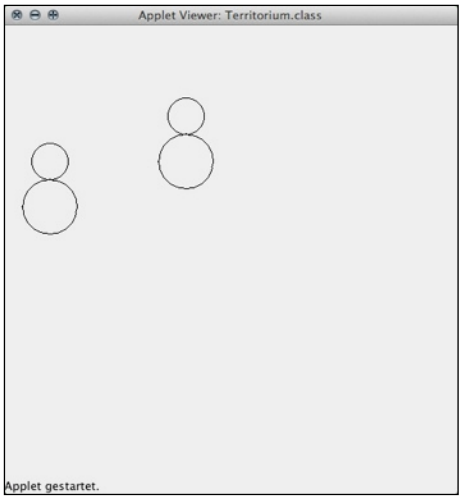
Betrachten Sie den folgenden Quelltext der Klasse **Roboter**:

```
import java.awt.*;

public class Roboter
{
    int xPos, yPos;

    public Roboter(int x, int y)
    {
        xPos = x;
        yPos = y;
    }

    public void zeigenVorn(Graphics g)
    {
        g.drawOval(xPos-20,yPos-70,40,40);
        g.drawOval(xPos-30,yPos-30,60,60);
    }

    public void zeigen(Graphics g)
    {
       zeigenVorn(g);
    }
}
```

Was wurde hier verändert gegenüber der ersten Version der Klasse **Roboter**?

In der "fertigen" Version unseres Roboter-Projektes soll sich der Roboter drehen können, und zwar jeweils um 90 Grad. Wenn wir das Applet starten, sehen wir den Roboter von vorn. Nach einem Befehl des Benutzers, beispielsweise durch Klicken eines Buttons, soll sich der Roboter aber um 90 Grad nach links oder rechts drehen können. Er muss dann von der rechten bzw. linken Seite zu sehen sein. Gibt der Benutzer den "drehen"-Befehl zweimal hintereinander, dreht sich der Roboter um 180 Grad, und er muss von hinten zu sehen sein. Wir benötigen also nicht nur eine, sondern vier verschiedene Methoden zum Zeigen des Roboters.

Für den Anfang belassen wir es aber bei der Darstellung von vorne, allerdings lagern wir den dazu nötigen Quelltext schon einmal in eine eigene Methode **zeigenVorn()** aus. Später werden wir die Klasse **Roboter** um drei analoge Methoden **zeigenRechts()**, **zeigenLinks()** und **zeigenHinten()** ergänzen. Welche dieser vier Methoden jetzt tatsächlich ausgeführt wird, wird in der Hauptmethode **zeigen()** festgelegt. Da es in dieser zweiten Version des Roboter-Projektes aber noch nichts zu entscheiden gibt - wir wollen den Roboter ja zunächst ausschließlich von vorne zeigen - wird in der **zeigen()**-Methode einfach die **zeigenVorn()**-Methode aufgerufen. Diese Methode malt in der zweiten Version übrigens nicht nur einen einfachen Kreis, sondern einen kleinen und einen großen Kreis. Der kleine Kreis könnte für den Kopf stehen, der große Kreis für den Rumpf.



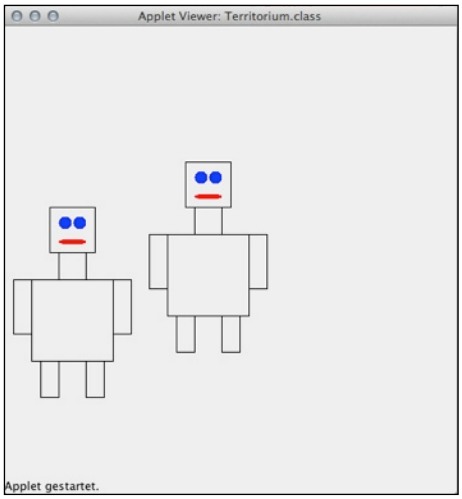
***6.1 - 2** Zwei Objekte der Klasse Roboter, Version 2*

Übung 6.1 - 1 (3 Punkte)

Auf Dauer wollen wir einen Roboter natürlich nicht durch einen Kreis darstellen, das wäre langweilig. Setzen Sie daher die in Folge 5 erlernten Java-Zeichenbefehle ein, um dem Roboter ein schöneres Antlitz zu geben. Falls Ihnen nichts besseres einfällt, schauen Sie sich die Abbildung unten an - sie zeigt einen einfachen Roboter, der aus sieben Rechtecken und drei Ovalen zusammengebaut wurde.

Ergänzungsübung (+2 Punkte)

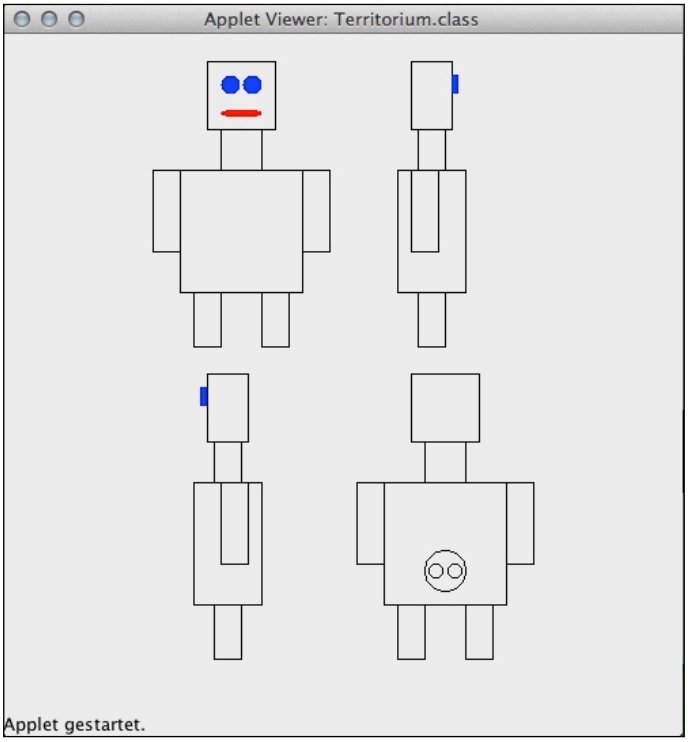
Für die Darstellung des Kopfes können Sie gern ein Objekt der Klasse **Gesicht** in den Roboter einbauen. Allerdings müssen Sie sich dann später etwas für die anderen drei zeigen-Methoden einfallen lassen, denn der Kopf muss ja auch von links, von rechts und von hinten zu sehen sein.



***6.1 - 3** Zwei Objekte der Klasse Roboter nach der Übung 6.1-1*

Schritt 4 - Ein Roboter mit vier Seiten

In der Übung 6.1 haben Sie den Roboter von *vorn* gezeichnet. Für unser Programm brauchen wir aber auch *seitliche* Ansichten des Roboters sowie eine Ansicht von *hinten*. Der Roboter soll sich nämlich um jeweils 90° nach links drehen können:



***6.1 - 4** Ein Applet mit vier verschiedenen Robotern in allen vier Ansichten*

Übung 6.1 - 2 (3 Punkte)

Schreiben Sie die drei Methoden **zeigenLinks()**, **zeigenRechts()** und **zeigenHinten()**, die genau das machen, was die Methodennamen besagen, nämlich den Roboter von links, von rechts und von hinten darzustellen. Lassen Sie dabei Ihre Phantasie spielen, aber achten Sie darauf, dass der Roboter beim Drehen nicht kleiner oder größer werden darf, und dass auch die Beine, die Arme, der Hals etc. ihre vertikale Position (auf der y-Achse beibehalten) müssen. Der Kopf darf beim Drehen also nicht plötzlich nach unten sacken, und der Hals oder die Arme dürfen beim Drehen nicht kürzer oder länger werden.

Bei der Rück-Ansicht des rechten Roboters kann man in Abbildung übrigens gut die Steckdose erkennen, mit der er wieder aufgeladen werden kann.

Als kleine Hilfe zeige ich Ihnen hier mal das Applet, das ich zum Testen der Klasse **Roboter** verwendet habe:

```
import java.awt.*;
import javax.swing.*;

public class Territorium extends JApplet
{
  Roboter robbi1, robbi2, robbi3, robbi4;

  public void init()
  {
      robbi1 = new Roboter(150,20);
      robbi2 = new Roboter(300,20);
      robbi3 = new Roboter(150,250);
      robbi4 = new Roboter(300,250);
  }

  public void paint(Graphics g)
  {
      robbi1.zeigen(g,1);
      robbi2.zeigen(g,2);
      robbi3.zeigen(g,3);
      robbi4.zeigen(g,4);
  }
}
```

Und hier noch eine Hilfe: Die Methode **zeigen()** der Klasse **Roboter**:

```
    public void zeigen(Graphics g, int richtung)
    {
       switch (richtung)
       {
          case  1: zeigenVorn(g); break;
          case  2: zeigenLinks(g); break;
          case  3: zeigenRechts(g); break;
          case  4: zeigenHinten(g);
       }
    }
```

Die Richtung, in die der Roboter schauen soll, wird mithilfe eines Parameters angegeben, der dann in der Methode **zeigen()** durch eine switch-Anweisung ausgewertet wird.

Schritt 5 - Einen Button erzeugen

Für eine kurze Zeit vergessen wir einmal die Klasse **Roboter** und wenden uns dem zweiten großen Thema der Folge 6 zu: Wie kann man ein Applet mithilfe von **Buttons** steuern? Dazu sind im Prinzip vier Schritte notwendig.

1. **Deklaration** des Buttons als Attribut des Applets
2. **Initialisierung** des Buttons in der **init()**-Methode des Applets
3. **Positionierung** und **Dimensionierung** des Buttons
4. **Aktivierung** des Buttons

5.1 Deklaration des Buttons

Beginnen wir mit der Deklaration des Buttons: Um einen Button zu erzeugen, müssen Sie das Applet mit einem Attribut ausstatten, das ein Objekt der Klasse **Button** ist:

```
public class Territorium extends JApplet
{
   Roboter robbi;
   Button btnVor;
   ...
```

5.2 Initialisierung des Buttons

Wie fast alle Initialisierungen erfolgt auch die Initialisierung der Buttons in der **init()**-Methode des Applets:

```
public void init()
{
   robbi = new Roboter(50,200);
   btnVor = new Button("Vor");
   ...
```

Bei dieser Initialisierung wird der Konstruktor der Klasse **Button** aufgerufen, der einen Parameter erwartet, nämlich die Beschriftung des Buttons.

5.3 Positionierung und Dimensionierung des Buttons

5.3.1 Vorbereitung

Wir wollen den Button an einer bestimmten Stelle des Applets *pixelgenau* unterbringen. Dazu müssen wir dem Applet zunächst mitteilen, *dass* wir das pixelgenaue Positionieren überhaupt aktivieren wollen. Normalerweise werden Buttons nämlich *mitten im Applet* dargestellt, und wenn ein Applet mehrere Buttons hat, werden diese einfach nebeneinander bzw. untereinander dargestellt.

Wollen wir unseren Buttons aber eigene Koordinaten zuweisen, so müssen wir den Befehl

```
setLayout(null);
```

in die **init()**-Methode des Applets schreiben.

Anmerkung: Hier gibt es an den Rechnern in meiner Schule immer wieder Probleme. Zu Hause bei den Schülern funktioniert alles so wie hier beschrieben, an den Rechnern in der Schule gibt es Fehlermeldungen oder falsche Ausgaben, weil hier andere Java-Runtime-Versionen installiert sind, teils auf unterschiedlichen Rechnern auch noch verschiedene Versionen. So hatte ich mir Java eigentlich nicht vorgestellt, war doch die "Plattformunabhängigkeit" eines der stärksten Argumente für Java überhaupt.

Wenn es also mit

```
setLayout(null);
```

nicht funktioniert, versuchen Sie es einmal mit dem langen Befehl

```
getContentPane().setLayout(null);
```

Es *könnte* sein, dass es dann funktioniert.

5.3.2 Positionierung und Dimensionierung

Nun können wir zur Positionierung und Dimensionierung (Bestimmung der Breite und Höhe) schreiten:

```
btnVor.setBounds(140,420,100,50);
```

Der Befehl **setBounds()** der Klasse **Button** erwartet vier Parameter. Die beiden ersten Parameter bestimmen die Position, die beiden anderen Parameter die Breite und Höhe des Buttons.

5.4 Hinzufügen des Buttons zur Komponentenliste

Ein wichtiger Befehl fehlt noch in der **init()**-Methode des Applets:

```
add(btnVor);
```

Um diesen letzten Befehl zu verstehen, müssen Sie sich vorstellen, dass jedes Applet eine Liste von **Komponenten** besitzt. Bei einem neuen Applet ist diese **Komponentenliste** zunächst leer. Wenn Sie aber einen oder mehrere Buttons oder andere Komponenten (zum Beispiel eine Listbox, ein Textfield, eine Checkbox und so weiter) haben wollen, müssen Sie diese der Komponentenliste hinzufügen, und zwar mit dem Applet-Befehl **add()** bzw. wenn es damit Probleme gibt, mit dem längeren Befehl **getContentPane().add()**.

5.5 Zusammenfassung

Betrachten wir jetzt noch einmal die vollständige **init()**-Methode des Applets:

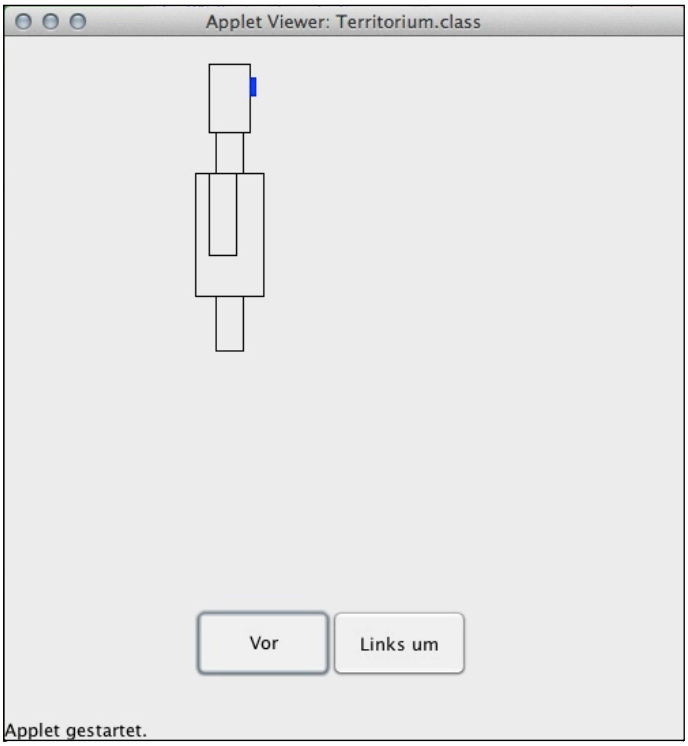
```
import java.awt.*;
import javax.swing.*;

public class Territorium extends JApplet
{
    Roboter robbi;
    Button btnVor;

    public void init()
    {
      robbi = new Roboter(150,20);
      btnVor = new Button("Vor");
      btnVor.setBounds(140,420,100,50);
      getContentPane().setLayout(null);
      getContentPane().add(btnVor);
    }

    public void paint(Graphics g)
    {
      robbi.zeigen(g,2);
    }
}
```

Die **paint()**-Methode ruft dann die zeigen()-Methode des Roboters auf, die ihrerseits die **zeigenLinks()**-Methode aufruft. Mit der Darstellung des Buttons hat die **paint()**-Methode aber nichts zu tun.

Wenn wir das Applet jetzt starten, sehen wir einen Button mit der Aufschrift "Vor" an der Position 140, 420 des Applets. Allerdings passiert noch nichts, wenn wir auf diesen Button klicken.

Übung 6.1 - 3 (3 Punkte)

Ergänzen Sie das Applet um einen zweiten Button. Dieser soll die Aufschrift "Links um" tragen und sich rechts oder links von dem Vor-Button auf gleicher Höhe befinden.



6.1 - 5 *Das Applet mit den zwei Buttons*

Schritt 6 - Aktivierung des Buttons

Noch passiert nichts, wenn Sie auf einen der beiden Buttons klicken. Wir müssen die Buttons erst aktivieren. Dazu sind zwei Schritte notwendig.

6.1 Einen ActionListener hinzufügen

Ergänzen Sie die ersten Zeilen des Applets wie folgt:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Territorium extends JApplet implements ActionListener
{
   Roboter robbi;
   ...
```

Außerdem müssen wir den jeweiligen Button mit diesem **ActionListener** verknüpfen. Dies geschieht in der **init()**-Methode des Applets. Ergänzen Sie die **init()**-Methode um die Zeile

```
btnLinks.addActionListener(this);
```

Verfahren Sie genau so für den anderen Button.

Hier zur Kontrolle noch einmal der gesamte Quelltext der **init()**-Methode:

```
public void init()
{
      robbi      = new Roboter();
      btnVor     = new Button("Vor");
      btnLinksUm = new Button("Links Um");
      add(btnVor);
      add(btnLinksUm);

      setLayout(null);
      btnLinksUm.setBounds(140,420,100,50);
      btnVor.setBounds(240,420,100,50);

      btnLinksUm.addActionListener(this);
      btnVor.addActionListener(this);
}
```

Wenn Sie das Applet jetzt kompilieren, erhalten Sie eine **Fehlermeldung**:

"Territorium is not abstract and does not override abstract method actionPerformed(java.awt.event.ActionEvent) in java.awt.event.ActionListener"

Diese Fehlermeldung erscheint, weil Sie das Applet mit einem **ActionListener** verbunden haben. Damit dieser **ActionListener** funktionieren kann, ***muss*** das Applet eine Methode **actionPerformed()** zur Verfügung stellen, welche die abstrakte Methode **actionPerformed()** der Klasse **ActionListener** überschreibt und mit konkretem Inhalt füllt.

6.2 Die fehlende Methode hinzufügen

Ergänzen Sie das Applet jetzt um folgende Methode:

```
public void actionPerformed(ActionEvent ereignis)
{
    if (ereignis.getSource() == btnLinksUm)
        robbi.linksUm();
    if (ereignis.getSource() == btnVor)
        robbi.vor();
    repaint();
}
```

Jetzt können Sie das Applet immer noch nicht kompilieren, obwohl das Applet selbst keinen einzigen Fehler mehr enthält. Die Klasse **Roboter** stellt nämlich weder die Methode **linksUm()** noch die Methode **vor()** zur Verfügung. Das ist aber schnell gemacht. Gemäß dem Prinzip der Top-Down-Programmierung statten wir die Klasse **Roboter** zunächst mit zwei leeren Methoden aus, damit wir das Applet kompilieren können:

```
public void vor()
{
  // später mit Inhalt füllen
}

public void linksUm()
{
   // später mit Inhalt füllen
}
```

Das Applet lässt sich jetzt kompilieren, allerdings passiert immer noch nichts, wenn wir auf einen der Buttons klicken. Das liegt natürlich daran, dass die **vor()**- und **linksUm()**-Methode der Klasse **Roboter** noch nicht mit Inhalt gefüllt sind.

Zunächst beschäftigen wir uns aber noch ein wenig mit der Methode **actionPerformed()**.

6.3 Die Methode actionPerformed()

Diese Methode sorgt dafür, dass das Applet auf bestimmte Ereignisse reagiert. Ein solches Ereignis kann beispielsweise das Drücken der linken Maustaste, das Antippen der Taste 'A' oder das Anklicken des LinksUm-Buttons sein. Wird ein Java-Applet mit einem ActionListener verbunden

```
public class Territorium extends JApplet implements ActionListener
```

so ***muss*** das Applet zwingend eine Methode **actionPerformed()** zur Verfügung stellen - andernfalls gibt es beim Kompilieren eine Fehlermeldung. Und diese Methode ***muss*** zwingend einen Parameter der Klasse **ActionEvent** haben.

Abiturienten achten bitte darauf, dass dieses Objekt nicht in dem Applet *erzeugt* wird, sondern der Klasse als Parameter *übergeben* wird. Es handelt sich streng genommen also um eine KENNT-Beziehung zwischen den Klassen **Applet** und **ActionEvent** und nicht um eine HAT-Beziehung (siehe "Beziehungen zwischen Objekten").

6.4 Die Methode getSource()

Für uns ist jetzt eine Methode der Klasse **ActionEvent** wichtig: **getSource()**. Mithilfe dieser Methode kann ermittelt werden, welche Komponente des Applets die Quelle des Ereignisse ist - mit anderen Worten: *welcher* Button geklickt wurde - der Button "Vor" oder der Button "Links Um".

Wenn der Button **btnLinksUm** angeklickt wurde, so wird die Methode **linksUm()** des Objektes **robbi** aufgerufen. Wurde dagegen der Button **btnVor** angeklickt, so wird die Methode **vor()** des Objektes **robbi** aufgerufen. So einfach ist die Sache.

Schritt 7 - Die linksUm()-Methode

Wenn der Roboter von vorn zu sehen ist und wenn er sich nach links umdreht, so soll anschließend seine rechte Seite zu sehen sein. Wenn er sich noch weiter nach links dreht (immer in 90°-Schritten), so ist seine Hinterseite zu sehen und so weiter.

Es ist also wichtig für die Klasse **Roboter**, die jeweilige Richtung zu kennen, in der sich das Roboter-Objekt befindet. Daher benötigen wir ein entsprechendes Attribut **richtung**, das wir im Konstruktor zunächst auf den Wert 1 (für "vorne") setzen.

```
import java.awt.*;

public class Roboter
{
    int xPos, yPos;
    int richtung;

    public Roboter(int x, int y)
    {
        xPos = x;
        yPos = y;
        richtung = 1; // von vorn;
    }
    ...
```

Wenn der Button **btnLinksUm** aufgerufen angeklickt wurde, soll sich zunächst die Richtung des Roboters ändern, danach muss die entsprechende **anzeigen()**-Methode aufgerufen werden.

Die Methode **linksUm()** gestaltet sich jetzt recht einfach; es muss nur die aktuelle Richtung geändert werden:

```
public void linksUm()
{
    richtung++;
    if (richtung == 5) richtung = 1;
}
```

Da es nur die vier Richtungen 1, 2, 3 und 4 gibt, muss die Richtung wieder auf den Wert 1 gesetzt werden, sobald der Roboter die Richtung 4 hat und sich nach links drehen soll.

Werfen wir nun einen Blick auf die entsprechende Stelle im Applet:

```
public void paint(Graphics g)
{
    robbi.anzeigen(g,2);
}
```

Hier hat sich noch nichts geändert. Der Roboter blickt nach rechts, und wenn man auf den **btnLinksUm**-Button klickt, passiert immer noch nichts. Woran liegt das?

Wenn der Benutzer auf den **btnLinksUm**-Button klickt, löst er in dem Applet ein Ereignis aus. Die Methode **actionPerformed()** erkennt,

a) es wurde ein Ereignis ausgelöst und

b) der **btnLinksUm**-Button wurde gedrückt.

Durch die Anweisung

```
if (ereignis.getSource() == btnLinksUm)
    robbi.linksUm();
```

wird nun die **linksUm()**-Methode der Klasse **Roboter** aufgerufen:

```
public void linksUm()
 {
    richtung++;
    if (richtung == 5) richtung = 1;
 }
```

Das Attribut **richtung** wird um 1 erhöht bzw. wieder auf den Wert 1 gesetzt, falls es vorher den Wert 4 hatte.

Nachdem die **actionPerformed()**-Methode die **linksUm()**-Methode aufgerufen hat, wird die **repaint()**-Methode ausgeführt. Diese Methode des Applets ruft nun die **paint()**-Methode auf, so dass das Applet neu gezeichnet wird. In der **paint()**-Methode wird nun die **zeigen()**-Methode des Roboters aufgerufen, und zwar mit dem Parameter 2:

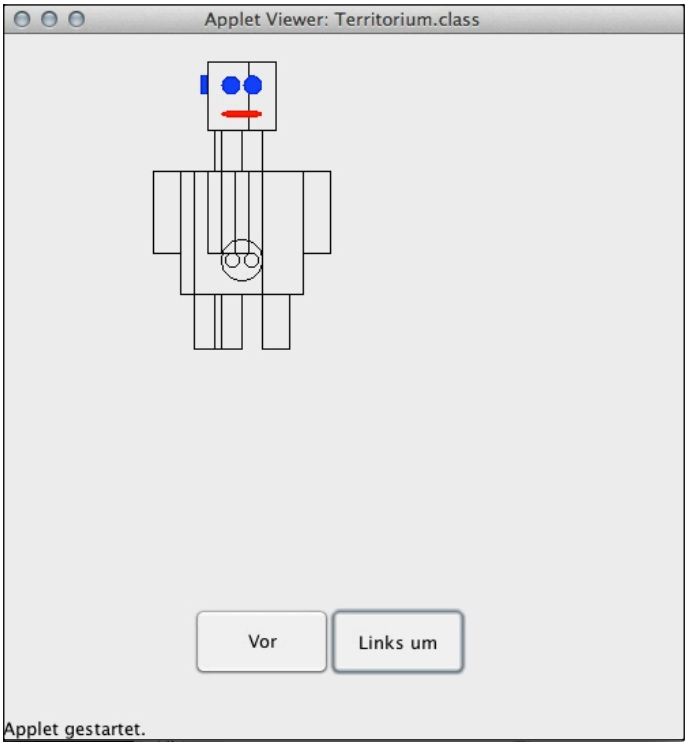
```
    public void zeigen(Graphics g, int richtung)
    {
       switch (richtung)
       {
          case  1: zeigenVorn(g); break;
          case  2: zeigenLinks(g); break;
          case  3: zeigenRechts(g); break;
          case  4: zeigenHinten(g);
       }
   }
```

Und genau hier liegt noch der Fehler. Der *Parameter* `richtung` ist jetzt völlig überflüssig geworden. Die **zeigen()**-Methode muss jetzt das *Attribut* **richtung** auswerten und danach entscheiden, welche der vier zeigen-Methoden ausgeführt wird:

```
    public void zeigen(Graphics g)
    {
       switch (richtung)
       {
          case  1: zeigenVorn(g); break;
          case  2: zeigenLinks(g); break;
          case  3: zeigenHinten(g); break;
          case  4: zeigenRechts(g);
       }
    }
```

Außerdem musste die Reihenfolge der Aufrufe etwas verändert werden. Wenn sich der Roboter immer um 90 Grad nach links dreht, ist er zunächst von vorn zu sehen, dann von links, dann von hinten und schließlich von rechts.

Jetzt passiert tatsächlich etwas, wenn der "Links Um"-Button betätigt wurde. Der nach vorne schauende Roboter wird überschrieben von einem nach links schauenden Roboter, der uns seine rechte Seite präsentiert. Wenn wir erneut den Button betätigen, wird die Rückseite des Roboters gezeichnet und so weiter. Allerdings werden die alten Ansichten des Roboters nicht gelöscht, und das ist nicht sehr schön:



***6.1 - 6** Die alten Ansichten werden nicht gelöscht*

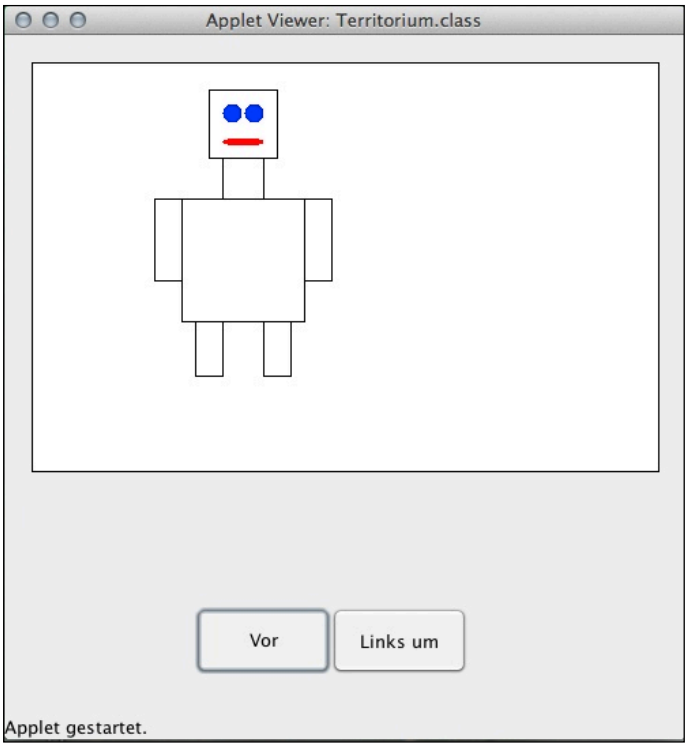
Schritt 8 - Endlich dreht sich der Roboter

Es gibt aber einen ganz einfachen "Trick", mit dem wir das Applet dazu bringen, das Gewünschte zu leisten: Jedes Mal, bevor wir die **paint()**-Methode des Roboters aufrufen, zeichnen wir ein weißes Rechteck in das Applet.

```
public void paint(Graphics g)
{
  g.setColor(Color.WHITE);
  g.fillRect(20,20,460,300);
  g.setColor(Color.BLACK);
  g.drawRect(20,20,460,300);
  robbi.zeigen(g);
}
```

Das Ganze sieht jetzt sehr schön aus, und der Roboter dreht sich wirklich, ohne dass die alte Ansicht überschrieben wird. Allerdings ist es nicht schön, dass der Kopf des Roboters direkt unter dem Rand des weißen Rechtecks "klebt". Das kann man aber in der **init()**-Methode des Applets leicht ändern, indem man die y-Koordinate des Roboter-Objekts auf den Wert 40 setzt:

```
robbi = new Roboter(150,40);
```



***6.1 - 7** Der Roboter wird nicht mehr übermalt*

Schritt 9 - Nach vorne gehen

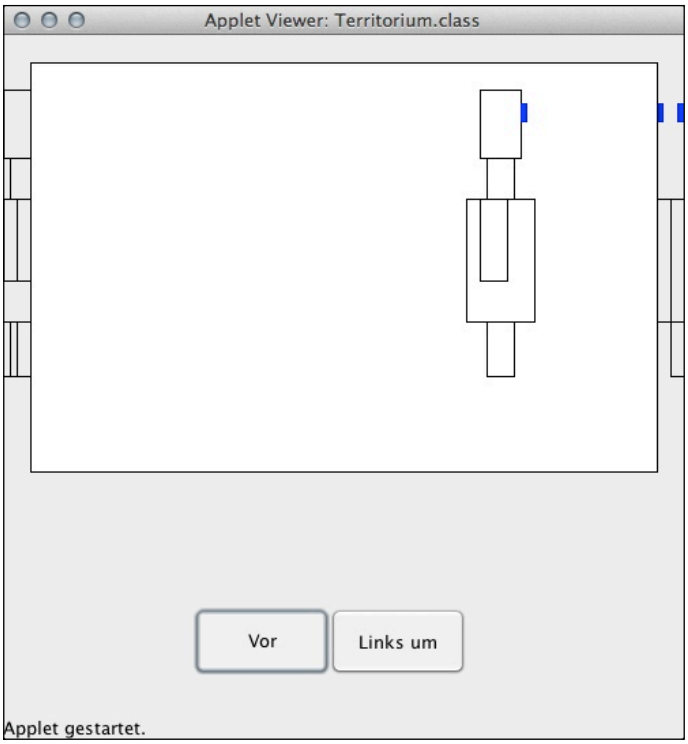
Wenn der "Vor"-Button aufgerufen wird, soll der Roboter einen Schritt nach vorne gehen. Wenn der Roboter gerade nach links oder nach rechts schaut, sollte das auch kein Problem sein; im Applet geht er dann beispielsweise 50 Pixel nach links oder nach rechts. Schwieriger ist es schon, wenn der Roboter nach vorne oder nach hinten schaut, dann müsste der auf uns zukommen und dabei (scheinbar) größer werden, oder er müsste von uns weggehen und dabei (scheinbar) kleiner werden.

Konzentrieren wir uns zunächst auf die leichtere Aufgabe: Der Roboter schaut nach links oder nach rechts.

```
public void vor()
{
    if (richtung == 2) xPos += 50;
    else if (richtung == 4) xPos -=50;
}
```

Mit dieser **vor()**-Methode erreichen wir das Gewünschte. Der Roboter geht 50 Pixel nach rechts, wenn er in die rechte Richtung schaut, und 50 Pixel nach links, wenn er nach links blickt.

Allerdings gibt es noch ein paar Unschönheiten, die es noch zu beseitigen gilt - womit wir auch gleich bei Ihren nächsten Aufgaben wären...



***6.1 - 8** Der Roboter kann das Appletfenster verlassen*

Schritt 10 - Übungen

Übung 6.1 - 4 (5 Punkte)

Verbessern Sie das Programm so, dass der Roboter nicht weitergehen kann, wenn er den linken bzw. rechten Rand des Applets erreicht hat. Gehen Sie davon aus, dass das Applet immer 500 Pixel breit ist.

Zusatzaufgabe (+ 2 Punkte)

Recherchieren Sie, mit welchem Befehl man die aktuelle Breite des Applets auslesen kann, und passen Sie das Programm so an, dass der Roboter das Applet nicht verlassen kann, egal wie breit es gerade ist.

Übung 6.1 - 5 (2 Punkte)

Ergänzen Sie das Applet und die Klasse **Roboter** um einen Button, der den Roboter um 90 Grad nach rechts dreht.

Unterrichtsvorhaben 2

6.2 Ein photorealistischer Roboter (I1,I2,I3 / A,M,I)

Anmerkung:

Diesen Unterrichtsvorhaben habe ich unverändert aus der vorherigen Version des Skriptes übernommen. Ich hatte leider noch keine Zeit, mich in die Tiefen der Graphikprogrammierung unter Java weiter einzuarbeiten. Ich bin sicher, dass es elegantere Methoden gibt, Bilder in einer Graphikklasse anzuzeigen, und wer Lust und Ahnung hat, kann mir ja mal eine einfachere, kürzere Version des Projektes zuschicken; ich würde diese dann unter Nennung Ihres Namens hier als Gastbeitrag veröffentlichen.

Schritt 1 - Einbinden von Bildern

Die Java-Befehle zum Zeichnen sind ja alle ganz nett, doch so richtig professionell sieht der Roboter damit nicht aus. Wie wäre es denn, wenn Sie die vier Ansichten mit Photoshop oder einem andern Zeichenprogramm malen und dann in das Java-Applet einbinden würden? Oder wenn Sie sich selbst von vier Seiten photographieren und die Bilder dann einbinden? Ich selbst habe mich sogar aus *acht* verschiedenen Richtungen photographiert, so dass sich mein Photo um 45° drehen kann, wenn ich auf den linksUm-Button klicke:

***6.2 - 1** Helmich x 8*

Die Hauptarbeit war nicht das Programmieren, sondern die Arbeit mit Photoshop. Erst mal musste ich alle acht Bilder freistellen, also vom lästigen Hintergrund befreien. Dann musste ich alle Bilder so skalieren, dass der Kopf, die Hose, die Hände und die Schuhe auf der gleichen y-Koordinate blei-

ben, wenn das Bild gewechselt wird. Das war dank der Ebenentechnik von Photoshop aber gar nicht so schwer, kostete aber trotzdem recht viel Zeit. Wenn Sie wollen, können Sie für Ihr Applet gern die acht Bilder dieser Seite verwenden, schöner wäre es aber, wenn Sie sich selbst photographieren würden.

Schritt 2 - Änderungen im Applet

Um ein Bild des Roboters zu zeigen, müssen Sie nur die Anzeigen-Methode(n) des Roboters ändern. Das Applet selbst kann weitgehend so bleiben, wie es war. Schauen Sie sich doch einmal die **init()**-Methode des von mir programmierten Applets an:

```
public void init()
{
   helmich = new Roboter(this);

   btnLinksUm = new Button("linksUm");
   btnVor     = new Button("vor");
   add(btnLinksUm);
   add(btnVor);

   setLayout(null);
   btnLinksUm.setBounds(10,460,160,30);
   btnLinksUm.addActionListener(this);
   btnVor.setBounds(210,460,160,30);
   btnVor.addActionListener(this);
}
```

Der besseren Übersicht wegen beschränke ich mich hier mal auf die einfache Version des Applets mit nur zwei Buttons. Auffällig ist nur die Zeile

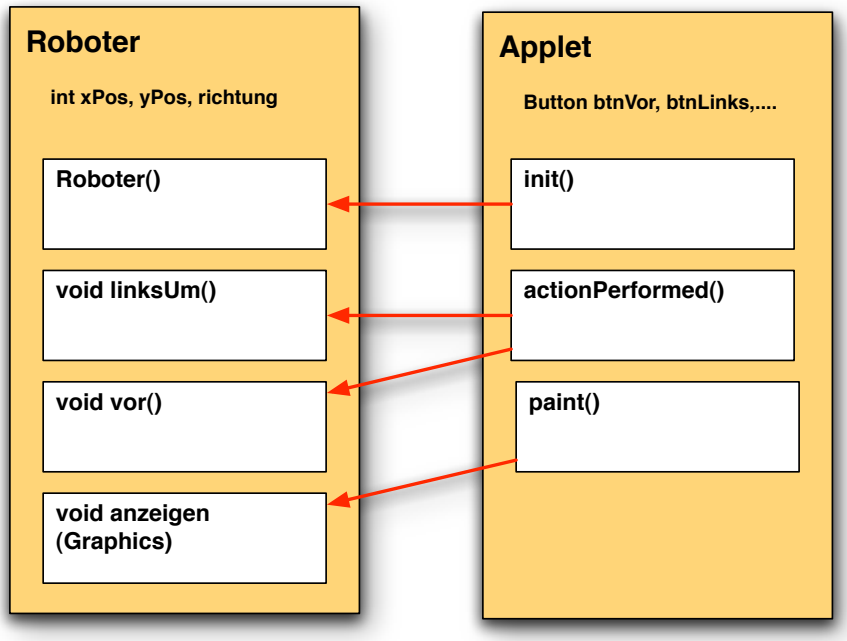
```
   helmich = new Roboter(this);
```

Dem Konstruktor der Klasse **Roboter** wird der Parameter `this` übergeben. Die Klasse **Roboter** - genauer gesagt, der Konstruktor - wird auf diese Weise mit dem aufrufenden Applet verknüpft, so dass der Konstruktor der Klasse **Roboter** auf das Applet zugreifen kann. Bisher war ja nur der umgekehrte Fall möglich, dass nämlich das Applet auf den Konstruktor der **Roboter**-Klasse zugreifen konnte. Startkoordinaten werden dieser Version des Roboters nicht mehr übergeben, der Roboter startet grundsätzlich links im Applet bei einer Position von x=50 und y=100.

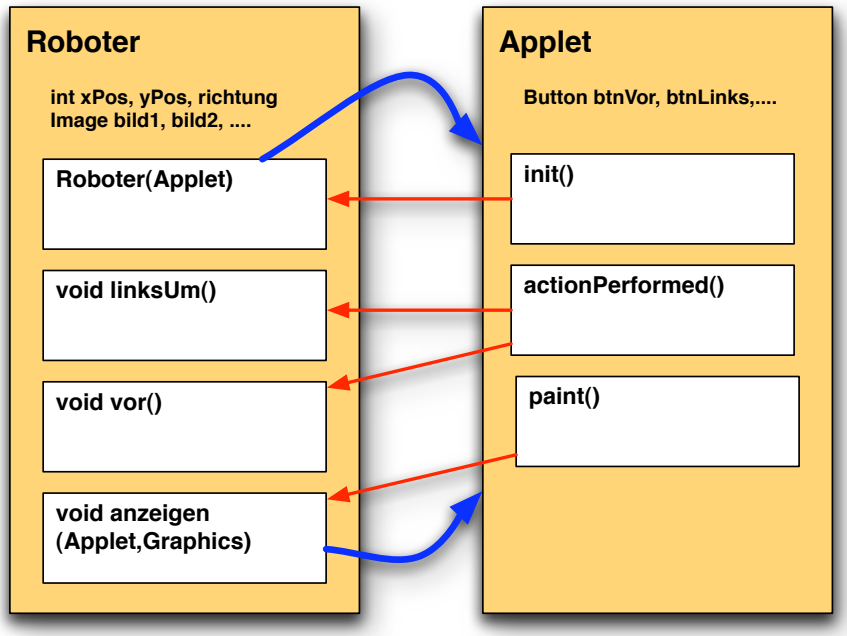
An der **actionPerformed()**-Methode musste ich überhaupt nichts ändern, die **paint()**-Methode dagegen enthält eine winzige Ergänzung:

```
public void paint(Graphics g)
{
   helmich.anzeigen(this,g);
}
```

Auch hier wird der **anzeigen()**-Methode ein Zeiger namens `this` auf das Applet übergeben. So kann auch diese Methode der Klasse **Roboter** direkt auf das Applet zugreifen. Warum das notwendig ist, werden Sie gleich sehen.



6.2 - 2 Beziehungen zwischen den Klassen bei der einfachen Roboter-Version



6.2 - 3 Beziehungen zwischen den Klassen bei der Roboter-Version mit Bildern

Hier ist das eben Gesagte noch einmal in Form von zwei Zeichnungen dargestellt. Abbildung 6.2 - 2 zeigt die bisherige Version, die **Applet**-Methoden konnten wohl auf die Methoden der **Roboter**-Objekte zugreifen, aber nicht umgekehrt. Abbildung 6.2 - 3 zeigt, wie es sich jetzt verhält: Bestimmte Methoden der Klasse **Roboter** können auch auf das **Applet** zugreifen.

Schritt 3 - Anpassung der Klasse Roboter

Hier der (gekürzte) Quelltext der Klasse **Roboter**, so wie ich ihn bei meinen Vorarbeiten zu diesem Skript programmiert habe:

```
import java.awt.*;
import java.applet.*;

public class Roboter
{
   Image bild1, bild2, bild3, bild4;
   Image bild5, bild6, bild7, bild8;

   int richtung;
   int xPos, zPos;

   public Roboter(Applet app)
   {
     bild1 = app.getImage(app.getCodeBase(),"1.jpg");
     bild2 = app.getImage(app.getCodeBase(),"2.jpg");
         und so weiter...

      bild8 = app.getImage(app.getCodeBase(),"8.jpg");

     richtung = 1;
     xPos = 50; zPos =  0;
   }

   public void linksUm()
   { ... }

   public void vor()
   { ... }

   public void anzeigen(Applet app, Graphics g)
   {
     if (richtung == 1)
        g.drawImage(bild1,xPos+zPos/2,50,168-zPos,400-zPos*2,app);
     else if (richtung == 2)
        g.drawImage(bild2,xPos+zPos/2,50,168-zPos,400-zPos*2,app);
     else
        und so weiter…

     else
       g.drawImage(bild8,xPos+zPos/2,50,168-zPos,400-zPos*2,app);
    }
}
```

Es taucht also eine neue Klasse auf, nämlich **Image**. Die Klasse **Roboter** enthält dann acht Objekte **bild1**, **bild2** etc. dieser Klasse. Ein Profi-Programmierer hätte hier natürlich einen aus acht Elementen bestehenden Array von Image-Objekten angelegt, aber Arrays werden ja erst in der nächsten Folge behandelt, so dass ich hier auf acht Einzelvariablen zurückgegriffen habe.

Schritt 4 - Die Änderungen verstehen

4.1 Der Konstruktor - Erzeugen von Bildern

Schauen wir uns noch einmal die ersten Zeilen des Konstruktors an:

```
public Roboter(Applet app)
{
   bild1 = app.getImage(app.getCodeBase(),"1.jpg");
   bild2 = app.getImage(app.getCodeBase(),"2.jpg");
   und so weiter...
```

Warum müssen wir dem Konstruktor einen Parameter `app` übergeben, der ja ein Objekt der Klasse **Applet** ist? Dazu betrachten wir die Anweisung in der ersten Quelltextzeile des Konstruktors:

```
    bild1 = app.getImage(app.getCodeBase(),"1.jpg");
```

Das Objekt **bild1** gehört zur Klasse **Image**. Um ein **Image**-Objekt zu erzeugen, müsste man normalerweise den Konstruktor der Klasse **Image** mit den entsprechenden Parametern aufrufen:

```
bild1 = new Image(...)
```

Das wird hier aber nicht gemacht. Statt dessen wird eine Methode **getImage()** der Klasse **Applet** aufgerufen. Dieser Methode werden zwei Parameter übergeben, nämlich erstens eine Angabe, in *welchem Verzeichnis* das Applet zu finden ist, und zweitens den *Dateinamen* bzw. *Pfad* des Bildes.

Der Pfad für das Applet wird durch den **Applet**-Befehl **getCodeBase()** automatisch ermittelt; man schreibt also einfach `app.getCodeBase()` als Aufruf-Parameter in die Klammern. Den Namen des Bildes muss man allerdings selbst angeben. Sollte sich das Bild in einem Unterverzeichnis wie z.B. "bilder" befinden, so muss man das Unterverzeichnis mit angeben, etwa so:

```
    bild1 = app.getImage(app.getCodeBase(),"bilder/1.jpg");
```

Achten Sie darauf, dass hier normale Schrägstriche (slash) verwendet werden und nicht wie bei Windows üblich umgekehrte Schrägstriche (backslash).

Kommen wir jetzt auf die eingangs gestellte Frage zurück, wieso der Konstruktor als Parameter ein Objekt der Klasse **Applet** benötigt? Innerhalb des Konstruktors wird auf Methoden der Klasse **Applet** zugegriffen, nämlich auf **getImage()** und auf **getCodeBase()**. Dazu muss dem Konstruktor aber Zugang zu dem Applet gewährt werden, das den Konstruktor aufruft. Genau das ist die Aufgabe des Parameters `app`. Er verknüpft den Konstruktor mit dem Applet, das den Konstruktor aufruft.

4.2 Die anzeigen()-Methode

Die nächste Änderung finden Sie in der **anzeigen()**-Methode der Klasse **Roboter**. Das Bild wird mit dem Befehl **drawImage()** der Klasse **Graphics** gezeichnet.

```
g.drawImage(bild8,xPos+zPos/2,50,168-zPos,400-zPos*2,app);
```

Diese Methode erwartet folgende Parameter:

- Parameter 1: Ein Objekt der Klasse **Image**, also das Bild, das gezeichnet werden soll.
- Parameter 2 und 3: Die Koordinaten der linken oberen Ecke des Bildes.
- Parameter 4 und 5: Die Breite und die Höhe des Bildes.
- Parameter 6: Ein Zeiger auf das **Applet**, in dem das Bild angezeigt werden soll.

Damit hat **anzeigen()** das gleiche Problem wie der Konstruktor von **Roboter**: Die Methode muss Zugriff auf das Applet haben, welches **anzeigen()** aufruft. Denn der letzte Parameter von **drawImage()** ist ein Objekt der Klasse **Applet**.

Falls Sie vorhaben, den bildlich dargestellten Roboter auch auf der z-Achse wandern zu lassen, so sind die Parameter 4 und 5 recht hilfreich. Wenn das Bild zum Beispiel 200 mal 300 Pixel groß ist, so erhalten Sie mit einer angegebenen Breite von 100 und einer angegebenen Höhe von 150 Pixeln ein nur halb so großes Bild. Die **drawImage()**-Methode *skaliert* das Bild automatisch. Auf das richtige *Seitenverhältnis* müssen Sie allerdings selbst achten.

Übung 6.2 - 1 (8 Punkte)

Programmieren Sie ein Applet, das wie oben beschrieben vier verschiedene Bilder des Roboters verwendet. Bei Betätigung des "Links Um" - Buttons dreht sich der Roboter um 90 Grad.

Es folgen nun noch ein paar Ergänzungsaufgaben für Experten und Expertinnen. Suchen Sie sich je nach Zeit und Leistungsvermögen eine, zwei, drei oder sogar alle vier Aufgaben aus der folgenden Liste aus. Die erste Experten-Aufgabe müssten eigentlich noch alle schaffen.

Ergänzungsaufgaben 6.2 - 2 für Experten (14 Punkte)

1. Verwenden Sie nicht vier, sondern *acht* verschiedene Bilder des Roboters aus acht verschiedenen Richtungen. Bei Betätigung des "Links Um" - Buttons dreht sich der Roboter dann um 45 Grad (2 Zusatzpunkte).
2. Der Roboter soll sich nicht nur nach links und rechts, sondern auch nach vorn und hinten bewegen können. Dabei soll der Roboter größer bzw. kleiner werden (3 Zusatzpunkte).
3. Verwenden Sie pro Richtung nicht ein Bild des Roboters, sondern zwei oder drei verschiedene, um die Bewegungen beim Gehen zu simulieren (3 Zusatzpunkte).
4. Verwenden Sie Photos von sich selbst statt die Bilder von mir. Bearbeiten Sie die Photos mit einem Bildbearbeitungsprogramm, so dass der Hintergrund einheitlich aussieht (nach Möglichkeit weiß) und dass die Person beim Drehen nicht kleiner oder größer wird (Oberkante Kopf, Unterkante Füße, Höhe des Gürtels müssen beim Drehen gleich bleiben) (6 weitere Punkte).

Und nun noch etwas für ganz besondere Spezialisten:

Ergänzungsaufgabe 6.2 - 3 für Super-Experten (4 Punkte)

Erkundigen Sie sich in Java-Büchern oder im Internet, wie man ein Java-Applet mithilfe der Tastatur steuern kann. Bauen Sie dann eine solche Tastatur-Steuerung in Ihr Applet ein. Sie könnten den Roboter dann zum Beispiel durch die Pfeil-Tasten in alle vier Richtungen gehen lassen, mit SHIFT-Pfeil-Links bzw. SHIFT-Pfeil-Rechts könnten Sie den Roboter drehen etc.

Folge 7 - Arrays

Unterrichtsvorhaben

7.1 Einfache Arrays (I1,I2,I3,I5 / A,M,I)

Schritt 1 - Einführendes Beispiel

```
public class Liste
{
    int z1, z2, z3, z4;

    public Liste()
    {
      z1 = 1; z2 = 4; z3 = 9; z4 = 16;
    }

    public void ausgeben()
    {
      System.out.println("1 hoch 2" + " = " + z1);
      System.out.println("2 hoch 2" + " = " + z2);
      System.out.println("3 hoch 2" + " = " + z3);
      System.out.println("4 hoch 2" + " = " + z4);
    }
}
```

Offensichtlich werden hier die Zahlen 1, 4, 9 und 16 erzeugt und angezeigt. Stellen Sie sich nun vor, Sie müssen das Programm so erweitern, dass die ersten 100 Quadratzahlen erzeugt und ausgegeben werden. Das wäre dann eine immense Tipparbeit für Sie, selbst wenn Sie wie ein Weltmeister mit Copy und Paste umgehen können. Aber es geht einfacher:

```
public class Liste
{
    int[] zahl;

    public Liste()
    {
      zahl = new int[100];
      for (int i=0; i<100; i++) zahl[i] = i*i;
    }

    public void ausgeben()
    {
      for (int k=0; k<100; k++)
        System.out.println("Zahl "+k+" = "+zahl[k]);
    }
}
```

Obwohl hier 100 Quadratzahlen erzeugt und ausgegeben werden, ist das Programm kaum länger als die erste Version mit den vier Zahlen. Wie kommt das? **Die zweite Version der Klasse Liste verwendet einen Array!**

Schritt 2 - Was ist eigentlich ein Array?

Ein Array ist eine **endliche Menge völlig gleichartiger Variablen.** Unser Array aus dem obigen Programm besteht aus 100 int-Variablen. Genau so gut kann man einen Array erzeugen, der aus 200 double-Variablen oder 2000 String-Variablen besteht.

Schritt 3 - Wie deklariert man einen Array?

Ein Array wird ähnlich wie ein normales Attribut deklariert. Vergleichen wir einmal die Deklaration einer einzelnen int-Variable und eines Arrays aus lauter int-Variablen:

```
int eineZahl;       // einzelne Variable
int[] achtZahlen;   // Array aus int-Zahlen
```

Der Array heißt zwar **achtZahlen**, die Arraygröße ist jedoch noch nicht festgelegt; das geschieht erst bei der **Initialisierung** des Arrays (s.u.). Auf die gleiche Weise könnte man auch einen Array aus double-Zahlen oder Strings deklarieren:

```
double[] zwanzigZahlen;
String[] textseite;
```

Schritt 4 - Was passiert bei der Deklaration?

Bei der Deklaration teilt man dem Rechner bzw. dem Java-System mit, *dass* man einen Array haben möchte, und wie der Array heißen soll. Nähere Angaben, vor allem wie viele Elemente der Array haben soll, hat man noch nicht gemacht.

Nach der Deklaration

```
int[] achtZahlen;
```

hat man zwar eine Variable erzeugt, die "zehnZahlen" heißt, aber in der Variable steht noch nichts drin, sie ist leer.

Schritt 5 - Wie initialisiert man einen Array?

Bei der Initialisierung wird die *Zahl* der gewünschten Array-Elemente festgelegt.

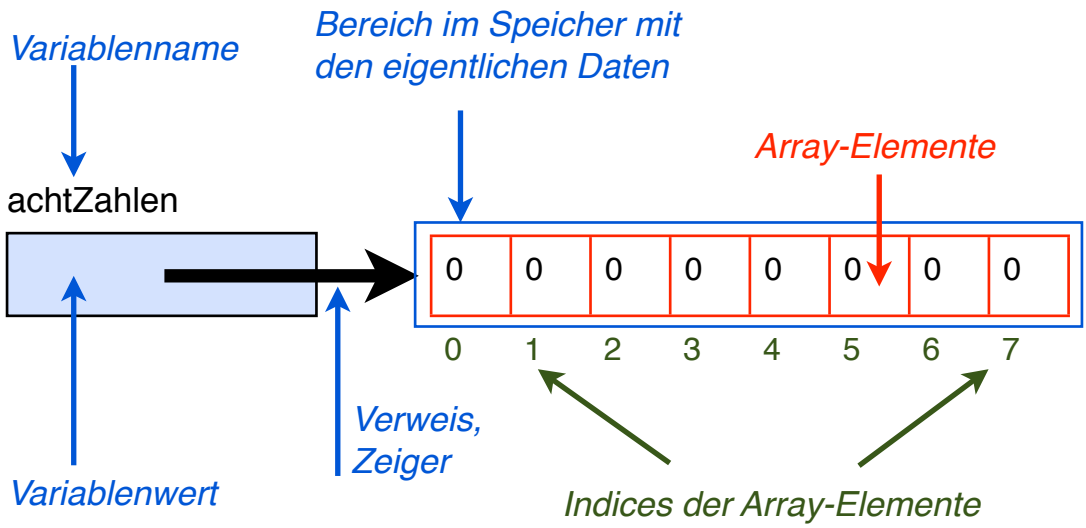
```
achtZahlen = new int[8];
```

Offensichtlich will man hier zehn int-Zahlen verwenden. Genau so gut hätte man aber schreiben können:

```
achtZahlen = new int[42];
```

Jetzt hätte man einen Array erzeugt, der 42 int-Zahl speichern kann, aber offensichtlich einen völlig falschen Namen hat. Dieses Beispiel zeigt mal wieder, wie wichtig doch die Wahl eines korrekten Variablen-, Attribut- oder Methodennamens ist.

Schritt 6 - Was passiert bei der Initialisierung eines Arrays?



7.1-1 Organisation eines Arrays

Die Array-Variable **achtZahlen** hat keinen "richtigen" Wert, sondern verweist auf einen Bereich im Speicher, in dem sich die eigentlichen Daten befinden, die **Array-Elemente**. In der Zeichnung werden diese Array-Elemente durch die roten Kästchen symbolisiert. Jede int-Zahl hat nach der Initialisierung den **default-Wert** `0`.

Mithilfe der **Indices** (Singular: Index) kann man auf jedes einzelne Array-Element direkt zugreifen. **Achten Sie darauf, dass der Index des ersten Array-Elementes immer den Wert `0` hat.**

Initialisierung eines Arrays

Die Initialisierung eines Arrays erzeugt einen Bereich im Arbeitsspeicher, in dem eine bestimmte Anzahl von Variablen gleichen Typs untergebracht werden kann. Direkt nach der Initialisierung stehen die default-Werte in den Array-Elementen (bei Zahlen `0`, bei Objekten `null`).

Syntax der Array-Initialisierung

```
bezeichner = new Datentyp[Anzahl];
```

Beispiele:

```
zahl = new int[12];
temperatur = new double[60];
inventar = new Gegenstand[7];
```

Man kann auch Arrays erzeugen, deren Elemente Objekte einer anderen Klasse sind:

```
Gegenstand[] inventar;
inventar = new Gegenstand[7];
```

erzeugt einen Array aus sieben Objekten der Klasse **Gegenstand**.

Schritt 7 - Wie weist man Array-Elementen Werte zu?

Betrachten wir zunächst ein Beispiel.

```
achtZahlen[0] = 1;
achtZahlen[1] = 4;
achtZahlen[2] = 9;
```

Zur Zuweisung eines Wertes an ein Array-Element benötigt man die *Adresse* dieses Elementes. Dazu dient der **Index**.

Zuweisung von Werten an Array-Elemente

Auf jedes Array-Element kann direkt mithilfe des Index zugegriffen werden. Das erste Array-Element hat immer den Index 0.

Syntax der Zuweisung

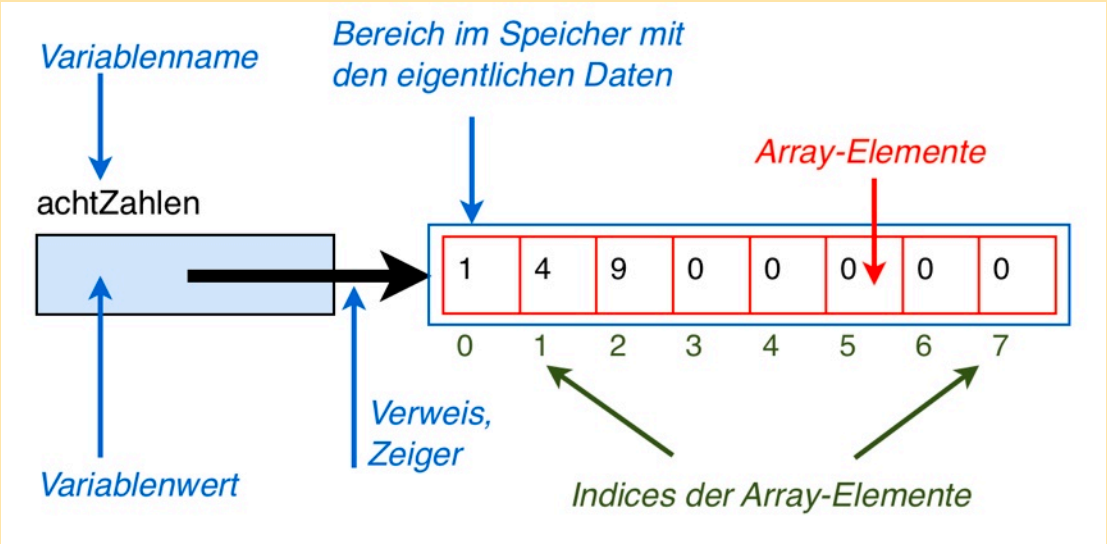
```
bezeichner[index] = Wert;
```

Beispiele:

```
zahl[0] = 13;
temperatur[12] = 27.3;
inventar[2] = new Gegenstand("Hammer",0,45,2);
for (int k=0; k<max; k++) liste[k] = k*k+3*k-2;
```

Das vorletzte Beispiel zeigt, wie man einem Element eines Objekt-Arrays einen Wert zuweist. Das letzte Beispiel zeigt wieder den Einsatz einer for-Schleife.

Indices sind relative Adressen



Die Adresse des gesamten Arrays ist in der Array-Variablen (hier: **achtZahlen**) gespeichert.
In diesem Speicherbereich (blau umrahmter Kasten) befinden sich die einzelnen Array-Elemente (rot umrahmt).

Jedes Array-Element kann dann durch seinen **Index** angesprochen werden, der auch als **relative Adresse** interpretiert werden kann.

Ein anschaulicheres Beispiel:

Die Array-Variable (hier: **achtZahlen**) könnte als Straßenname interpretiert werden, das Array-Element (hier zum Beispiel **achtZahlen[2]**) als Hausnummer. Mit diesen beiden Angaben kann nun der Wert des Array-Elementes ermittelt oder verändert werden. Mit dem Befehl

```
achtZahlen[2] = 9;
```

wird der Wert des Array-Elementes verändert, und mit einer Abfrage wie

```
if (achtZahlen[2] % 2 == 0)
```

wird lesend auf den Wert zugegriffen (in diesem Beispiel wird geprüft, ob die int-Zahl durch 2 teilbar ist).

Schritt 8: Arrays und for-Schleifen

Wenn man 100 Array-Elementen einen Wert zuweisen will, könnte das in eine ziemlich intensive Tipparbeit ausarten. Falls sich die Werte der 100 Elemente irgendwie aus den Indices berechnen lassen, kann man die Wertzuweisung mithilfe einer for-Schleife in *einer* Zeile Quelltext automatisch erledigen lassen. Betrachten Sie dazu das folgende Beispiel:

```
for (int i=0; i<100; i++) zahl[i] = i*i;
```

Hier wird 100 Array-Elementen je eine Quadratzahl zugewiesen. Das erste Element mit dem Index 0 bekommt den Wert 0, das zweite Element den Wert 1, das dritte Element den Wert 4 und so weiter. Das letzte Element mit dem Index 99 bekommt dann den Wert $99^2 = 9801$ zugewiesen.

Bei der Verwendung von solchen for-Schleifen muss man aber aufpassen, dass man keinen "Array-out-of-bounds"-Fehler programmiert. Betrachten Sie dazu folgenden *falschen* Quelltext:

```
for (int i=1; i<=100; i++) zahl[i] = i*i;
```

So falsch sieht der Quelltext doch gar nicht aus, meinen Sie?

Fehler 1:

Das erste Array-Element bekommt keinen Wert zugewiesen. Denn das erste Array-Element hat den Index 0, was man leicht vergisst. Die for-Schleife beginnt aber mit dem Index 1, der ja das *zweite* Array-Element anspricht. Dieser Fehler ist allerdings nicht besonders gravierend; er bringt das Programm zumindest nicht zum Absturz.

Fehler 2:

Dieser Fehler ist gravierender als der erste. Angenommen, die for-Schleife ist so oft durchlaufen worden, dass die Laufvariable `i` jetzt den Wert 100 hat.

Die Bedingung `i <= 100` ist dann noch erfüllt.

Nun hat der Array zwar genau 100 Elemente, das letzte Element hat aber den Index 99. Ein Element mit dem Index 100 existiert nicht.

Es wird also in der for-Schleife versucht, auf ein Array-Element zuzugreifen, das außerhalb des Arrays liegt. Daher auch der Name dieses Fehlers - "array-out-of-bounds".

Nicht immer eignen sich Schleifen für Arrays

Nicht immer können Schleifen bei der Zuweisung von Werten an Arrays eingesetzt werden. Stellen Sie sich vor, Sie haben ein Programm geschrieben, mit dem Sie Ihre Noten in den einzelnen Schulfächern verwalten wollen. Dann müssen Sie jede Note einzeln eingeben, es sei denn, Sie stehen überall auf 1 (oder 5, was wir nicht hoffen wollen).

Schritt 9 - Wie greift man auf die Werte von Array-Elementen zu?

Zunächst ein paar Beispiele:

```
System.out.println(zahl[3]);                // 1
int ergebnis = zahl[2];                     // 2
int summe = zahl[0] + zahl[1] + zahl[2];    // 3
if (zahl[1] < 0) zahl[1] = 0;               // 4
inventar[4].ausgeben();                     // 5
```

Wie man leicht sieht, können Array-Elemente wie normale Attribute oder Variablen behandelt werden. Man muss aber immer den Index des Elementes angeben, welches man gerade meint.

1. Ein Array-Element kann mit dem **println()**-Befehl ausgegeben werden,
2. den Wert eines Array-Elements kann man einer anderen Variablen zuweisen,
3. man kann mehrere Element gleichzeitig auswerten,
4. man kann innerhalb von if-Abfragen, while-Schleifen etc. auf den Wert bestimmter Array-Elemente zugreifen,
5. in einem Objekt-Array kann auf die einzelnen Objekte zugegriffen werden, man kann sogar Methoden dieser Objekte aufrufen und ausführen lassen. Hier wird die **ausgeben()**-Methode des fünften **Gegenstand**-Objektes des Arrays **inventar** aufgerufen.

Zugriff auf Werte von Array-Elementen

Auf jedes einzelne Array-Element kann *lesend* zugegriffen werden. Das gewünschte Array-Element wird durch Angabe des Index spezifiziert und verhält sich dann wie eine normale Variable.

Beispiele:

```
int y = zahl[0];
System.out.println(temperatur[12]);
if (zahlenliste[4] >= 20) bedingung = true;
inventar[3].beschaedigen(20);
```

Das letzte Beispiel zeigt, dass Objekte, die in einem Array gespeichert worden sind, wie ganz normale Objekte behandelt werden können. Man kann also beispielsweise die **beschaedigen()**-Methode des vierten Objektes eines Arrays aufrufen.

Schritt 10 - Übungen mit Lösungen

Übung 7.1 - 1

Schreiben Sie Java-Code, der die beiden ersten Elemente des int-Arrays **zahl** vertauscht.

Lösung:

Leider gibt es in Java nicht die Möglichkeit eines *direkten* Tauschs, sondern wir müssen eine temporäre lokale Variable einsetzen, die als Zwischenspeicher fungiert:

```
int temp = zahl[0];
zahl[0] = zahl[1];
zahl[1] = temp;
```

Übung 7.1 - 2

Schreiben Sie Java-Code, der die beiden ersten Elemente des int-Arrays nur dann vertauscht, wenn das erste Element größer ist als das zweite Element.

Lösung:

```
if (zahl[0] > zahl[1])
{
   int temp = zahl[0];
   zahl[0] = zahl[1];
   zahl[1] = temp;
}
```

Übung 7.1 - 3

Schreiben Sie Java-Code, der einen int-Array der Größe 20 untersucht und die Anzahl der geraden int-Zahlen ermittelt.

Lösung

```
int gerade = 0;
for (int i=0; i<20; i++)
   if (zahl[i] % 2 == 0) gerade++;

System.out.println("Zahl der geraden Elemente = ",gerade);
```

Schritt 11 - Übungen

Übung 7.1 - 4 (2 Punkte)

Schreiben Sie eine sondierende Methode, welche die Summe der Elemente eines int-Arrays zurück liefert:

```
public int summe()
{ ... }
```

Übung 7.1 - 5 (4 Punkte)

Schreiben Sie eine sondierende Methode, die die kleinste Zahl liefert, die in einem int-Array enthalten ist.:

```
public int kleinsteZahl()
{ ... }
```

Übung 7.1 - 6 (4 Punkte)

Schreiben Sie eine manipulierende Methode, die die Werte eines Array "umdreht":

```
public void dreheArrayUm()
{ ... }
```

aus

```
5 - 12 -19 - 4 - 13 - 6
```

würde dann beispielsweise

```
6 - 13- 4 - 19 - 12 - 5
```

Nochmaliges Aufrufen von dreheArrayUm() stellt dann den ursprünglichen Array wieder her.

Schritt 12 - Zufallszahlen

Betrachten Sie die folgenden Quelltext:

```
import java.util.Random;

public class Liste
{
    private int[] zahl;

    public Liste()
    {
      zahl = new int[100];
    }

    public void erzeugen()
    {
      Random zufall = new Random();
      for (int i=0; i<100; i++)
        zahl[i] = zufall.nextInt(1000);
    }
    ...
}
```

Um **Zufallszahlen** zu erzeugen, benötigt man zuerst ein Objekt der Klasse **Random**. Die Klasse **Random** stellt den **Zufallszahlen-Generator** zur Verfügung. Damit unsere Java-Klasse **Liste** überhaupt auf die Klasse **Random** und ihre Methoden zugreifen kann, muss man die Java-Bibliothek **java.util.Random** mit dem `import`-Befehl am Anfang des Quelltextes einbinden.

In der Methode **erzeugen()** wird ein Objekt **zufall** der Klasse **Random** deklariert und initialisiert; der Konstruktor von **Random** erwartet keine Parameter.

Dann kommt eine for-Schleife, die Laufvariable `i` wird von 0 auf 99 hochgezählt. Eine Zufallszahl zwischen 0 und 999 erhält man durch Aufruf der Methode **nextInt()** der Klasse **Random** mit dem Parameter 1000. Will man Zufallszahlen zwischen 1 und 1000 haben, so muss man schreiben:

```
    zahl[i] = zufall.nextInt(1000)+1;
```

Merke: Zufallszahlen

Um Zufallszahlen vom Typ int zu erzeugen, benötigt man die Java-Klasse **Random**. Diese wird durch die Bibliothek `java.util.Random` zur Verfügung gestellt, die man in den Quelltext mit `import` einbinden muss.

Im Quelltext ist dann ein Objekt der Klasse **Random** zu deklarieren und zu initialisieren.

Dann kann mit der Methode **nextInt(max)** eine Zufallszahl zwischen 0 und max-1 erzeugt werden.

Schritt 13 - Weitere Übungen

Übung 7.1 - 7 (2 Punkte)

Schreiben Sie für die Klasse **Liste** aus Schritt 12 eine sondierende Methode

```
public boolean enthaelt(int x)
```

welche den Wert `true` zurück liefert, wenn die Zahl `x` in dem Array enthalten ist.

Übung 7.1 - 8 (2 Punkte)

Schreiben Sie für die Klasse **Liste** eine weitere sondierende Methode

```
public boolean enthaelt(int x1, int x2)
```

welche den Wert `true` zurück liefert, wenn *beide* Zahlen in dem Array enthalten sind.

Eine Java-Klasse kann durchaus zwei oder drei Methoden mit dem gleichen Namen besitzen. Allerdings müssen sich diese Methoden dann in ihren **Parameterlisten** unterscheiden. Die erste Methode **enthaelt()** besitzt *einen* int-Parameter, die zweite Methode dagegen *zwei* int-Parameter. Daher kann beim Aufruf der Methode **enthaelt()** leicht entschieden werden, welche der beiden Methoden eigentlich gemeint ist.

Merke: Overloading

Eine Klasse kann mehrere Methoden mit dem gleichen Namen zur Verfügung stellen. Diese Methoden müssen aber unterschiedliche Parameterlisten haben.

Beispiel:

```
public boolean enthaelt(int zahl);
public boolean enthaelt(int zahl1, int zahl2);
```

Übung 7.1 - 9 (3 Punkte)

Schreiben Sie für die Klasse **Liste** eine manipulierende Methode

```
public void mische()
```

welche die Array-Elemente gründlich durchmischt. Diese Methode testen Sie am besten nicht mit 100 Zufallszahlen (die sind ja schon gut gemischt), sondern mit 100 aufsteigenden Zahlen . Aus der Reihe

```
1 - 2- 3 - 4- ... - 98 - 99 - 100
```

wird nach dem Mischen beispielsweise die Reihe

```
25 - 4 - 2 - 36 - ... - 9 - 81 - 64
```

Unterrichtsvorhaben

7.2 Objekt-Arrays (I1 / M,I)(fakultativ für EF)

Schritt 1 - Klasse Gegenstand

Für ein Abenteuerspiel wurde eine Klasse **Gegenstand** entwickelt. Bei einem solchen Gegenstand kann es sich beispielsweise um ein Schwert oder eine andere Waffe handeln, oder um einen Helm, einen Brustpanzer, einen Heiltrank oder Ähnliches, wie es eben in Abenteuerspielen üblich ist.

Hier ein erster Entwurf der Klasse **Gegenstand** - natürlich noch nicht vollständig, aber für unsere Zwecke reicht er zunächst aus.

```
public class Gegenstand
{
    String name;
    int angriffswert, verteidigungswert, goldwert, zustand;

    public Gegenstand(String n, int a, int v, int g)
    {
       name              = n;
       angriffswert      = a;
       verteidigungswert = v;
       goldwert          = g;
       zustand           = 100; // Prozent
    }

    public void anzeigen()
    {
        System.out.println("Bezeichnung: "+"\t\t"+name);
        System.out.println("Angriffswert: "+"\t\t"+angriffswert);
        System.out.println("Verteidigungswert: "+"\t"+verteidigungswert);
        System.out.println("Goldwert: "+"\t\t"+goldwert);
        System.out.println("Zustand: "+"\t\t"+zustand + "%");
    }
}
```

In der **anzeigen()**-Methode wird übrigens reichlich Gebrauch von Tabulatoren gemacht. Wenn Sie in den Ausgabe-String des **System.out.println()**-Befehls den Teilstring `"\t"` einbauen, so wird an dieser Stelle ein **Tabulator** eingefügt. Mithilfe solcher Tabulatoren kann man die Darstellung in der Konsole wesentlich übersichtlicher gestalten als mithilfe von Leerzeichen.

In dem diesem Unterrichtsvorhaben wollen wir einen Array aus Objekten der Klasse **Gegenstand** implementieren und diesen Array in eine neue Klasse **Inventar** einbetten. Ein Held hat ja meistens einen Rucksack dabei, in dem er viele Gegenstände aufbewahrt. Die Klasse **Inventar** soll es ermöglichen, Rucksäcke, Schatzkisten, Schränke und andere "Behälter" zu erzeugen, in denen Gegenstände aufbewahrt werden können.

Schritt 2 - Klasse Inventar

```
public class Inventar
{
    Gegenstand[] g;

    public Inventar()
    {
       g = new Gegenstand[12];

       g[ 0] = new Gegenstand("Hammer",20,3,567);
       g[ 1] = new Gegenstand("Helm",0,13,230);
       g[ 2] = new Gegenstand("Stiefel",4,10,177);
       g[ 3] = new Gegenstand("Schwert",40,3,1234);
       g[ 4] = new Gegenstand("Messer",20,1,12);
       g[ 5] = new Gegenstand("Knieschoner",1,10,40);
       g[ 6] = new Gegenstand("Schulterpanzer",0,12,564);
       g[ 7] = new Gegenstand("Panzerkragen",0,13,120);
       g[ 8] = new Gegenstand("Heiltrank",0,0,12);
       g[ 9] = new Gegenstand("Krafttrank",4,0,34);
       g[10] = new Gegenstand("Taschentücher",0,0,2);
       g[11] = new Gegenstand("Inhalator",4,0,5);
    }

    public void anzeigen()
    {
       for (int i=0; i<=11; i++)
       {
          g[i].anzeigen();
          System.out.println("----------------------------");
       }
    }
}
```

Hier sehen Sie die Klasse **Inventar** mit zwölf Gegenständen. Da jeder Gegenstand einen eigenen Namen und eigene Werte hat, kann man leider keine for-Schleife für die Wertzuweisungen einsetzen. Jeder Gegenstand muss einzeln initialisiert werden.

Bei der **anzeigen()**-Methode greifen wir aber wieder auf eine bewährte Methode zurück; wir setzen nämlich eine for-Schleife ein, die nacheinander die **anzeigen()**-Methoden der einzelnen Array-Elemente **g[i]** aufruft.

Schritt 3 - Objekt-Arrays verstehen

Deklaration des Objekt-Arrays

Schauen wir uns den Quelltext von **Inventar** genauer an. Am Anfang wird der Array **g** deklariert:

```
public class Inventar
{
   Gegenstand[] g;
```

Mit den eckigen `[]`-Klammern teilen wir dem Programm mit, dass wir nicht ein einzelnes Objekt der Klasse **Gegenstand** benötigen, sondern *viele Objekte* dieser Klasse. Die genaue Anzahl der Objekte wird in der **Deklaration** aber noch nicht festgelegt.

Initialisierung des Arrays

Die **Initialisierung** eines Objekt-Arrays erfolgt im Konstruktor der aufrufenden Klasse:

```
    public Inventar()
    {
        g = new Gegenstand[12];
```

Hier sagen wir dem Programm, dass wir genau zwölf Objekte der Klasse **Gegenstand** anlegen wollen. Konkret erzeugt werden die Objekte allerdings noch nicht, sondern es werden lediglich zwölf Referenzen (Zeiger) erzeugt, die später auf zwölf Objekte der Klasse **Gegenstand** verweisen.

Erzeugung der Array-Elemente

```
        g[ 0] = new Gegenstand("Hammer",20,3,567);
        g[ 1] = new Gegenstand("Helm",0,13,230);
        u.s.w.
```

Hier müssen wir wirklich zwölf Zeilen hinschreiben, weil ja jeder Gegenstand einen Namen und drei Werte benötigt, die sich der Computer nicht einfach ausdenken kann. Mit `g[0]` sprechen wir den *ersten* Gegenstand an.

Zugriff auf die Array-Elemente

Mit einer simplen for-Schleife und nur zwei Zeilen Code können alle zwölf Gegenstände ausgegeben werden:

```
        for (int i=0; i<=11; i++)
        {
            g[i].anzeigen();
            System.out.println("--------------------------");
        }
```

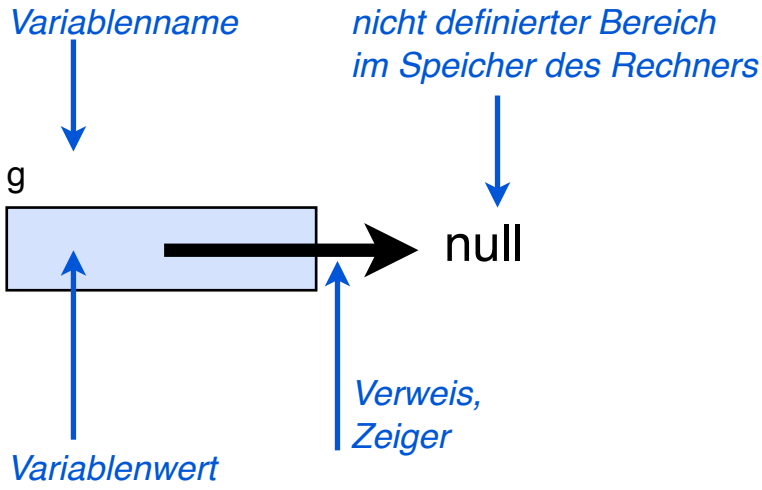
Als Index für das jeweils auszugebende Array-Element verwenden wir hier die Laufvariable `i` der for-Schleife. Am Anfang hat `i` den Wert 0, was ja dem ersten Array-Element entspricht. Also wird die **anzeigen()**-Methode des Objektes `g[0]` aufgerufen. Beim nächsten Schleifendurchgang hat `i` schon den Wert 1, und es wird `g[1]`, also das zweite Array-Element ausgegeben. Und so weiter, bis schließlich `g[11]`, das zwölfte Array-Element, ausgegeben wird.

7.3 Arrays gründlich verstehen

Mithilfe einer Reihe von kleinen Zeichnungen soll Ihnen nun verdeutlicht werden, wie solche Objekt-Arrays intern aufgebaut sind.

Deklaration einer Array-Variablen

Die Array-Variable **g** in den letzten Quelltexten sieht direkt nach der Deklaration so aus:

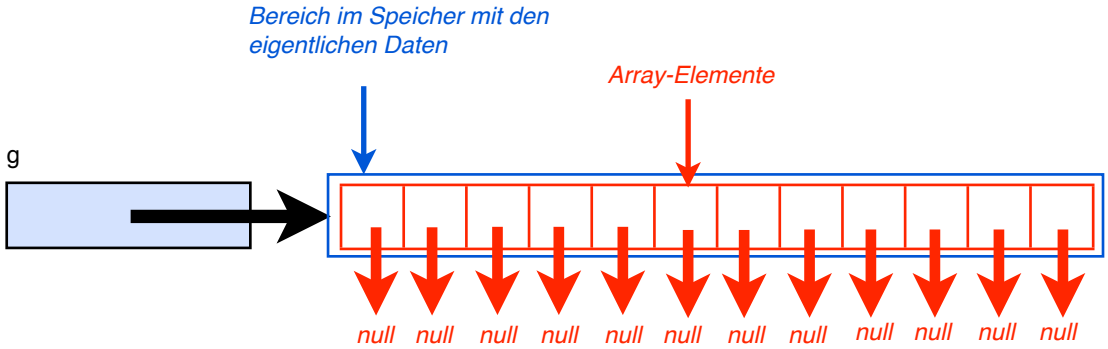


7.3 - 1 Darstellung eines Objekt-Arrays nach der Deklaration

Die Variable **g** enthält keine herkömmlichen Daten, sondern zeigt auf einen Bereich im Speicher des Rechners. Solche Variablen werden daher auch als **Zeiger** oder **Pointer** bezeichnet.

Initialisierung einer Array-Variablen

Bei der Initialisierung wird die Zahl der gewünschten Array-Elemente festgelegt. In unserem Quelltextbeispiel waren die Array-Elemente zwölf Objekte der Klasse **Gegenstand**:



7.3 - 2 Der Array nach der Initialisierung

Die zwölf Array-Elemente (im Bild die roten quadratischen Kästchen) sind nach der Initialisierung zwar schon vorhanden, sie enthalten aber noch keine Daten. Genauer gesagt, handelt es sich bei den zwölf Array-Elementen nicht um Objekte, sondern um *Zeiger auf Objekte*. Nach der Initialisierung

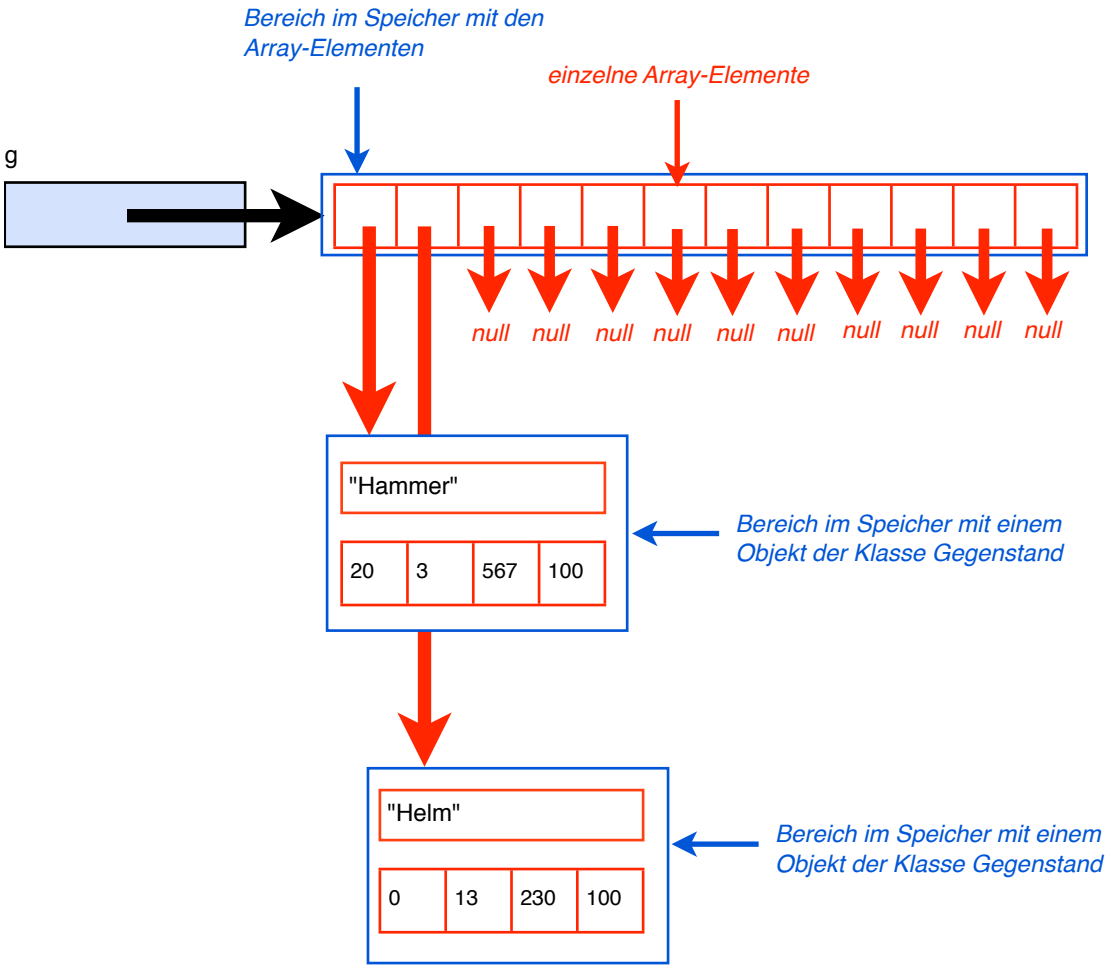
verweisen diese Zeiger allerdings noch auf keinen konkreten Bereich im Speicher des Rechners. Man sagt, die Zeiger haben den Wert `null`.

Erzeugung der Objekte

Wenn die ersten zwei Gegenstände erzeugt worden sind, also nach Ausführung der Zeilen

```
g[ 0] = new Gegenstand("Hammer",20,3,567);
g[ 1] = new Gegenstand("Helm",0,13,230);
```

sieht das Bild schon etwas anders aus:

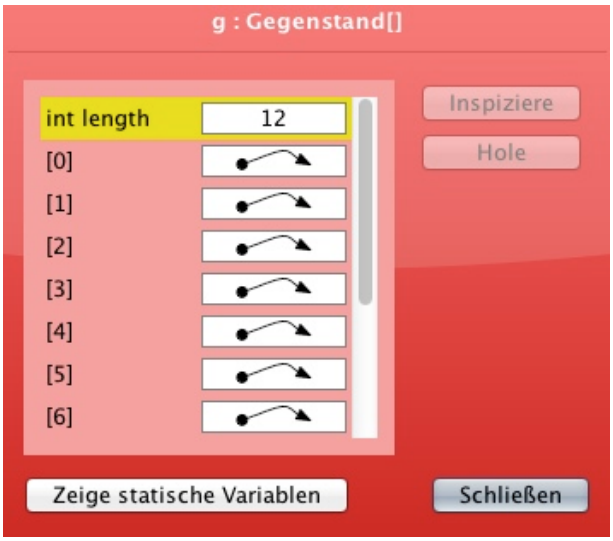


7.3 - 3 Der Array nach der Erzeugung von zwei Gegenständen

Wenn Sie sich selbst davon überzeugen wollen, dass der Array **g** tatsächlich so aufgebaut ist, wie im Text beschrieben, starten Sie doch einfach mal den Objektinspektor von BlueJ und klicken Sie auf die Pfeile, die Sie sehen.

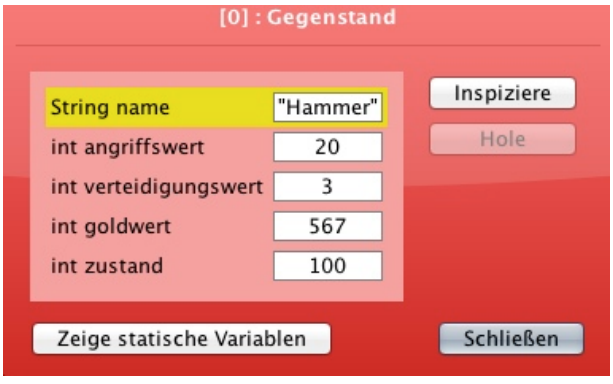
7.3 - 4 Inspektion eines Inventar-Objektes

Nun klicken Sie auf den Pfeil hinter `Gegenstand[] g`:



7.3 - 5 Inspektion des Arrays g

Hier sehen Sie die zwölf Array-Elemente - es handelt sich wieder um Zeiger. Wenn Sie dann auf den ersten Zeiger `[0]` in dem Array klicken, erhalten Sie Informationen über den jeweiligen Gegenstand:



7.3 - 6 Inspektion eines Array-Elements

Workshop (fakultativ)

7.4 Mit bunten Kreisen spielen (I1,I2 / A,M,I)

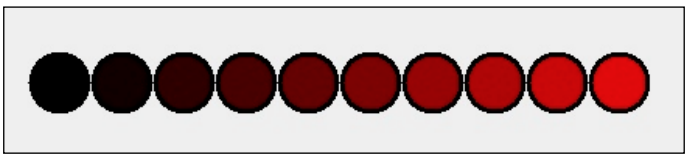
In diesem fakultativen Workshop wollen wir mit Objekt-Arrays arbeiten. Sie erinnern sich an die Klasse **Kreis**, die wir in der Folge 5.3 konstruiert hatten. **Kreis**-Objekte sind Graphik-Objekte, die beispielsweise von einem Applet aufgerufen und auf der Zeichenfläche dargestellt werden können.

Schritt 1

Betrachten Sie das folgende Java-Applet:

```
import java.awt.*;
import javax.swing.*;

public class MeinApplet extends JApplet
{
   Kreis[] k;

   public void init()
   {
      k = new Kreis[10];

      for (int i=0; i<10; i++)
         k[i] = new Kreis(50+i*40,150,20,new Color(i*25,0,0),Color.BLACK,3);

   }
   public void paint(Graphics g)
   {
      for (int i=0; i<10; i++)
         k[i].paint(g);
   }
}
```

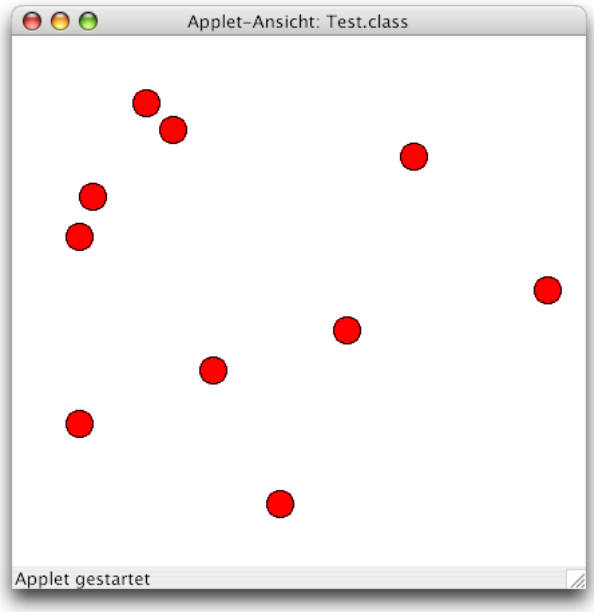
So sieht das Ergebnis der **paint()**-Methode aus:



7.4-1 Die zehn erzeugten Kreise

Schritt 2 - Zufallskreise

Betrachten Sie das folgende Applet



7.4 - 2 Ein Java-Applet mit zehn Städten

Die Abbildung soll zehn Städte eines Landkreises darstellen. Jede Stadt hat eine X-Koordinate und eine Y-Koordinate. Wie kann man *zehn* solcher Städte speichern?

Es gibt hier mehrere Möglichkeiten, von denen einige recht elegant sind, andere weniger. Da wir aber schon die Klasse **Kreis** entwickelt haben und gesehen haben, wie man **Kreis**-Objekte in ein Applet einbindet, sollte die Implementation eines entsprechenden Applets für Sie kein Problem sein. Das einzig Neue an dieser Aufgabe ist die Verwendung eines Zufallsgenerators für die Festlegung der Positionen der zehn Städte.

Übung 7.4 - 1 (3 Punkte)

Verändern Sie die **init()**-Methode des Applets aus Schritt 1 so, dass zehn gleich große rote Kreise mit schwarzem Rand zufällig über das Applet verteilt werden. Die Kreise müssen sich vollständig innerhalb des Applets befinden, den Rand des Applets dürfen sie nicht berühren.

Schritt 3 - Bewegliche Kreise

Wenn Sie die Folge 6 mit der "Robotersteuerung" durchgearbeitet haben, sollte die nächste Übung für Sie überhaupt kein Problem sein. Sie müssen nur wissen, wie man einen Button in ein Applet einbaut und durch welchen Trick man die Ausgabe eines Applets komplett bzw. teilweise löschen kann - der Button sollte besser nicht gelöscht werden.

Übung 7.4 - 2 (4 Punkte)

Ergänzen Sie das Applet aus Übung 7.4 - 1 um einen Button, den Sie mit "Bewegen" oder "Move" beschriften. Jedes Mal, wenn der Benutzer auf diesen Button klickt, sollen für jeden der zehn Kreise die Koordinaten neu berechnet werden. Und zwar soll der x-Wert um einen kleinen Zufallsbetrag zwischen -10 und +10 erhöht werden, und der y-Wert ebenfalls. Nach der Neuberechnung der Koordinaten sollen die zehn Kreise erneut gezeichnet werden (Löschen der alten Kreise nicht vergessen!).

Übung 7.4 - 3 (4 Punkte)

Verändern Sie das Applet aus Übung 7.4 - 2 so, dass die Kreise das Applet nicht verlassen können, wenn der Button angeklickt wurde. Wenn ein Kreis den Rand des Applets erreicht, soll er seine zufälligen Bewegungen nicht einstellen, sondern sich weiter bewegen - allerdings nicht über den Rand hinaus!

Lösungshinweis zur Übung 7.4 - 3

Angenommen, der Kreis mit dem Index 4 hat die x-Koordinate 20 und die y-Koordinate 160. Durch Klicken auf den "Move"-Button wird die x-Koordinate jetzt auf 4 und die y-Koordinate auf 170 verändert. Mit einem Radius von 10 würde der Kreis den linken Rand des Applets nicht nur berühren, sondern ein großer Teil des Kreises würde sogar außerhalb des Applets liegen. Das geht natürlich auf gar keinen Fall. Also muss für die x-Koordinate eine neue Zufallszahl ermittelt werden - und war so oft, bis der x-Wert in Ordnung ist.

Schritt 4 - Kreise, die sich vermehren

Wenn Sie einen Array angelegt haben, der 200 Objekte der Klasse **Kreis** enthält, zwingt Sie niemand, auch tatsächlich alle 200 Speicherplätze *sofort* mit Kreis-Objekten zu belegen.

```
import java.awt.*;
import javax.swing.*;

public class MeinApplet extends JApplet
{
   Kreis[] k;
   int belegt;

   public void init()
   {
      k = new Kreis[200];
      belegt = 10;
      for (int i=0; i<belegt; i++)
         k[i] = new Kreis(50+i*40,150,20,Color.RED,Color.BLACK,1);
   }

   public void paint(Graphics g)
   {
      for (int i=0; i<belegt; i++)
         k[i].paint(g);
   }
}
```

In diesem Applet zeigt das Attribut **k** auf eine Stelle im Arbeitsspeicher, an der bis zu 200 Objekte der Klasse **Kreis** gespeichert werden können. In der **init()**-Methode werden jedoch nur *zehn* Kreis-Objekte erzeugt und in dem Array gespeichert. Das Attribut **belegt** speichert dann die Zahl der *tatsächlich* belegten Array-Plätze.

Sie können nun jederzeit *neue* Array-Elemente erzeugen, also Objekte der Klasse **Kreis**, und diese in den noch freien Plätzen speichern. Sie müssen nur darauf achten, dass Sie dann auch den Wert von **belegt** entsprechend vergrößern.

```
public void weitererKreis()
{
    if (belegt < 200)
       k[belegt++] = new Kreis(150,250,20,Color.RED,Color.BLACK,1);
}
```

Diese Methode ergänzt den Array um einen weiteren Kreis und inkrementiert danach den Wert von **belegt**. Zuvor wird überprüft, ob **belegt** eventuell den maximalen Wert 200 erreicht hat. Dann wird nämlich nichts mehr gemacht.

Interessant ist hier die Tatsache, dass **belegt** sofort nach dem Zuweisen des neuen Elementes inkrementiert wird. Durch `k[belegt++]` wird zunächst das Element **k[belegt]** mit einem Objekt verknüpft, anschließend wird **belegt** inkrementiert.

Experten-Übung 7.4 - 4 (5 Punkte)

Nehmen Sie das Applet aus den letzten Übungen, verändern Sie es so, dass von den 200 möglichen Kreisen erst zehn belegt sind (Attribut **belegt** nicht vergessen). Beim Start des Applets werden also nur 10 rote Kreise angezeigt.

Beim Anklicken des Buttons sollen sich die Kreise bewegen (siehe Übung 7.4 - 2) und das Applet nicht verlassen können (siehe Übung 7.4 - 3).

Jetzt kommt die neue Aufgabe für die Experten unter Ihnen:

Immer dann, wenn sich zwei Kreise berühren (hier müssen Sie also eine *Kollisionsabfrage* einbauen), soll ein *neuer* Kreis entstehen und in den Array aufgenommen werden. Sie können die Kreise also als rote Käfer interpretieren, die sich gelegentlich vermehren.

Experten-Übung 7.4 - 5 (4 Punkte)

Ergänzen Sie das Applet aus Übung 7.4 - 4 derart, dass die neuen Kreise zunächst *kleiner* dargestellt werden als die alten Kreise. Bei jedem Klick auf den Button wachsen die kleinen Kreise (die Jungtiere) dann, bis sie den Wert der alten Kreise (Eltern) erreicht haben (2 Punkte).

Sorgen Sie weiterhin dafür, dass sich nur erwachsene Tiere vermehren können. Wenn also zwei Jungtiere oder ein Alttier und ein Jungtier zusammenstoßen, passiert nichts weiter (2 Punkte).

Superexperten-Übung 7.4 - 6 (6 Punkte)

Statten Sie das Applet mit *zwei* Arrays von Kreisobjekten aus: maximal 200 kleine grüne Kreise (Blattläuse) und maximal 25 größere rote Kreise (Marienkäfer). Die Marienkäfer sollen nun die Blattläuse fressen. Jedes Mal, wenn zwei erwachsene Marienkäfer zusammenstoßen, entsteht ein neuer junger Marienkäfer (siehe Übung 7.4 - 5). Jedes Mal, wenn zwei erwachsene Blattläuse zusammenstoßen, entsteht eine neue junge Blattlaus. Jedes Mal, wenn ein junger oder erwachsener Marienkäfer auf eine junge oder erwachsene Blattlaus stößt, wird die Blattlaus gefressen (aus dem Objekt-Array gelöscht oder als "gefressen" gekennzeichnet, so dass sie nicht mehr angezeigt wird).

Folge 8 -Sortierverfahren

8.1 Sortieren - Allgemeines (I2,I5 / A)

Sortieren ist einer der häufigsten Prozesse in der praktischen Informatik und eines der interessantesten Themen in der theoretischen Informatik. Warum ist das Sortieren überhaupt so wichtig, könnte man sich fragen. Der Grund dafür, dass in der Informatik ständig sortiert wird, liegt in den **Suchverfahren**. In einer sortierten Liste, Kartei, Bibliothek, CD-Sammlung etc. findet man das Gesuchte wesentlich schneller als in einer unsortierten Datensammlung.

Falls Ihnen das Gesagte noch nicht klar sein sollte - hier ein einfaches Anschauungs-Beispiel:

20	3	5	12	17	4	9	2	13

In dieser *unsortierten* Liste wollen wir die Zahl 4 suchen. Als Mensch können Sie sich diese neun Zahlen *gleichzeitig* anschauen und sehen sofort die 4 an der 6. Stelle der Liste.

Ein Computer kann das nicht. Er muss *von vorne* anfangen und *jede* Zahl einzeln untersuchen: Handelt es sich bei der ersten Zahl um die gesuchte Zahl? Nein! Handelt es sich bei der zweiten Zahl um die Suchzahl? Nein! Und so weiter. Bis er schließlich, nach sechs Vergleichen, bei der 4 landet.

Sortieren wir nun die obige Liste und suchen dann erneut nach der 4:

2	3	4	5	9	12	13	17	20

Bereits nach dem dritten Vergleich findet der Computer bzw. der Algorithmus die Suchzahl 4. Sie sehen also an diesem einfachen Beispiel, wie sinnvoll es ist, eine Datensammlung vor dem Suchen zu Sortieren.

Suchen wir in beiden Listen nach den Zahlen 1, 2, 3, ... 10, so benötigen wir für die Suche in der unsortierten Liste 71 Vergleiche, für die Suche in der sortierten Liste aber nur 37 Vergleiche. Die Einzelheiten hierzu finden Sie in dem Experten-Kasten auf der nächsten Seite.

Sie sehen jetzt, warum Sortierverfahren so besonders wichtig für die Informatik sind. Ist eine Datensammlung erst einmal sortiert, so ist das Suchen nach Daten oft kein großes Problem mehr.

Für Experten: Berechnung der Suchzeiten

Wir wollen in den beiden Listen (unsortiert / sortiert) nach den Zahlen 1, 2, ... , 10 suchen. Wie lange benötigen wir im Schnitt für das Suchen nach einer dieser Zahlen?

Durchschnittliche Suchzeit für die unsortierte Liste.

Um die Zahl 1 zu finden, benötigen wir 9 Vergleiche, denn wir müssen die Liste bis zum Ende durchsuchen, um sicher zu sein, dass die 1 gar nicht vorhanden ist. Die Zahl 2 finden wir nach 8 Vergleichen, die Zahl 3 nach 2 Vergleichen, die Zahl 4 nach 6 Vergleichen und die Zahl 5 nach 3 Vergleichen. Die 6, die 7 und die 8 finden wir gar nicht, mussten aber 9 Vergleiche durchführen, für die 9 brauchen wir 7 Vergleiche und für die 10 wieder 9 Vergleiche, weil sie gar nicht vorhanden ist.

Zählen wir nun einmal zusammen, dann kommen wir auf 9 + 8 + 2 + 6 + 3 + 9 + 9 + 9 + 7 + 9 = 71 Vergleiche für das Suchen nach den Zahlen 1 bis 10. Für eine einzelne Zahl sind das im Schnitt 7,1 Vergleiche.

Durchschnittliche Suchzeit für die sortierte Liste.

Um festzustellen, dass die 1 nicht in der Liste vorkommt, benötigen wir nur einen Vergleich, denn die erste Liste der Zahl ist die 2, also kann 1 nicht vorhanden sein. Die 2 finden wir nach 1 Vergleich, die 3 nach 2 Vergleichen, die 4 nach 3 Vergleichen, die 5 nach 4 Vergleichen. Die 6 ist nicht vorhanden, um das festzustellen, benötigen wir aber nicht 9 Vergleiche wie bei der unsortierten Liste, sondern nur 5 Vergleiche. Auch für die 7 und die 8 benötigen wir 5 Vergleiche. Auch die 9 finden wir nach 5 Vergleichen, und nach 6 Vergleichen wissen wir, dass die 10 nicht in der Liste vorkommt.

Wir kommen jetzt auf 1 + 1 + 2 + 3 + 4 + 5 + 5 + 5+ 5 + 6 = 37 Vergleiche, im Schnitt also auf 3,7 Vergleiche pro Suchzahl.

Fazit: Das Suchen in einer sortierten Liste geht also wesentlich schneller als das Suchen in einer unsortierten Liste.

Dabei haben wir noch nicht einmal ein optimales Suchverfahren für die sortierte Liste angewandt. Man hätte ja auch eine binäre Suche durchführen können, was die Suchzeit noch einmal stark reduziert hätte.

8.2 Die Klasse Liste (I1,I2,I3 / A,M,I)

Wir wollen uns in diesem Skript zunächst mit **einfachen Sortierverfahren** beschäftigen, und unsere Daten - in der Regel ganze Zahlen - werden auch nur nach *einem* Kriterium sortiert, nämlich nach dem Wert der Zahl.

Bei der Entwicklung der drei einfachen Sortierverfahren **Bubblesort**, **Selectionsort** und **Insertionsort** legen wir die Java-Klasse **Liste** zu Grunde, deren Quelltext Sie hier sehen:

```
import java.util.Random;

public class Liste
{
    public static final int MAX = 100; // Konstante!
    private int[] zahl;

    public Liste()
    {
        zahl = new int[MAX];
    }

    public void erzeugen()
    {
        Random zufall = new Random();
        for (int i=0; i<MAX; i++)
            zahl[i] = zufall.nextInt(10*MAX);
    }

    public void anzeigen()
    {
       for (int i=1; i<=MAX; i++)
       {
          System.out.print(zahl[i-1]+"\t");
          if (i % 10 == 0) System.out.println();
       }
    }
}
```

Die Zahl der Array-Elemente wurde hier als Konstante definiert.

Merke: Konstanten

Konstanten werden durch die Verwendung der Schlüsselworte `static` und `final` definiert. Üblicherweise (aber nicht zwingend) werden die Namen von Konstanten GROSS geschrieben.

Wenn Sie im Quelltext versuchen, den Wert einer Konstante zu verändern, liefert Ihnen der Compiler eine Fehlermeldung! Konstanten können nicht mehr verändert werden (Schlüsselwort `final`).

8.3 Der Bubblesort - ein einfaches Sortierverfahren (I2,I4 / A,D)

Schritt 1 - Das Verfahren kennen lernen

Beim Bubblesort werden immer zwei benachbarte Zahlen verglichen. Ist die linke Zahl größer als die rechte, so werden die beiden Zahlen vertauscht. Nach diesem Verfahren wird der gesamte Array durchsucht. Wenn der *erste Durchgang* beendet ist, befindet sich die *größte* Zahl des Arrays am Ende. Die anderen Zahlen des Arrays sind aber noch unsortiert, so dass weitere Durchgänge notwendig sind, bis alle Zahlen sortiert sind.

Wir wollen dieses einfache Sortierverfahren an einem kleinen Beispiel kennenlernen. Unser Array soll aus sechs int-Zahlen bestehen:

8	5	9	2	1	3

Durchgang 1

Zu Beginn werden die beiden ersten Zahlen miteinander verglichen - 8 und 5. Da die 8 größer ist als ihr rechter Nachbar, muss getauscht werden. Wir erhalten:

5	8	9	2	1	3

Nun überprüfen wir die 8 und ihren rechten Nachbarn, die 9. Da die 8 aber bereits kleiner ist als die 9, passiert nichts, und die nächsten beiden Elemente werden überprüft, die 9 und die 2. Hier muss wieder getauscht werden:

5	8	2	9	1	3

Nun werden die 9 und die 1 verglichen und getauscht:

5	8	2	1	9	3

Jetzt werden die beiden letzten Elemente verglichen und wieder getauscht:

5	8	2	1	3	9

Sie sehen selbst, dass der Array noch keineswegs sortiert ist. Der ganze beschriebene Vorgang war ja nur der *erste* von möglicherweise vielen Durchgängen. Am Ende dieses ersten Durchgangs ist eines allerdings sicher: Die *größte* Zahl des Arrays steht ganz hinten. Insofern ist der rechte Teil des Arrays bereits sortiert:

5	8	2	1	3	9
U					S

Dieser **sortierte Teil S** kann beim zweiten Durchgang ignoriert werden. Spielen wir auch diesen zweiten Durchgang noch einmal Schritt für Schritt durch.

Durchgang 2

5	8	2	1	3	9
U					S

Die beiden ersten Elemente des unsortierten Teils **U** werden verglichen. Da 5 < 8 ist, passiert nichts, und die beiden nächsten Elemente, 8 und 2, werden verglichen und getauscht:

5	2	8	1	3	9

Nun werden die beiden nächsten Elemente verglichen und vertauscht:

5	2	1	8	3	9

Dann die beiden nächsten Elemente:

5	2	1	3	8	9

Damit befindet sich die größte Zahl des noch unsortierten Teils, die 8, ganz rechts und gehört von nun an zum sortierten Teil des Arrays, der nicht mehr untersucht werden muss:

5	2	1	3	8	9
U				S	

Weitere Durchgänge

Wie viele Durchgänge brauchen wir, um den Array komplett zu sortieren? Im dritten Durchgang kommt die 5 an die Position 3:

2	1	3	5	8	9

Im vierten Durchgang ändert sich an der Reihenfolge der Zahlen nichts, aber der sortierte Teil ist gewachsen:

1	2	3	5	8	9

Wir sehen jetzt auf den ersten Blick, dass der Array sortiert ist. Der Algorithmus hat diese Kenntnis aber noch nicht. Daher ist ein fünfter Durchgang notwendig:

1	2	3	5	8	9

Der unsortierte Teil besteht nur noch aus einem Element, damit entfällt ein sechster Durchgang.

Wenn wir jetzt verallgemeinern, so können wir behaupten: **Wenn ein Array aus N Elementen besteht, so sind maximal N-1 Durchgänge notwendig, um ihn zu sortieren.**

Schritt 2 - Implementation des Verfahrens

Ergänzen Sie nun bitte den Quelltext der Klasse **Liste** um folgende zwei Methoden:

```
private void tausche(int index1, int index2)
{
    int temp = zahl[index1];
    zahl[index1] = zahl[index2];
    zahl[index2] = temp;
}

public void bubblesort()
{
   for (int durchgang = 1; durchgang < MAX; durchgang++)
      for (int index = 0; index < MAX-durchgang; index++)
         if (zahl[index] > zahl[index+1])
            tausche(index,index+1);
}
```

Normalerweise setzt man zwei ineinander verschachtelte for-Schleifen ein, wenn man einen Bubblesort implementieren möchte. Die innere for-Schleife durchläuft den Array von vorne nach hinten (bzw. von links nach rechts) und korrigiert eventuelle Fehlstellungen durch sofortiges Austauschen. Die äußere for-Schleife ist dafür verantwortlich, dass dieser Vorgang so oft aufgerufen wird, bis der gesamte Array sortiert ist.

Das Tauschen der Zahlen im Array wird hier durch die private Methode **tausche()** ausgeführt, die nach dem Ringtausch-Prinzip arbeitet.

Übung 8.3 - 1 (4 Punkte)

Der Bubblesort gilt als einer der *langsamsten* Sortieralgorithmen überhaupt. Recherchieren Sie, warum das so ist und tragen Sie Ihre Ergebnisse kurz vor (2 Punkte).

Vertiefen Sie Ihren Vortrag, indem Sie eine mathematische Formel entwickeln oder recherchieren, in der die Laufzeit eines Bubblesort in Abhängigkeit von der Zahl der zu sortierenden Elemente hergeleitet wird. Erläutern Sie das Zustandekommen dieser Formel (2 Punkte zusätzlich).

Übung 8.3 - 2A (12 Punkte)

Ermitteln Sie experimentell, inwiefern die Sortierzeit von der Anzahl der Array-Elemente abhängt. Dazu müssen Sie im Internet recherchieren, wie man in Java die aktuelle Systemzeit auslesen und sinnvoll auswerten kann. In einer der vielen Java-Klassen gibt es bestimmte eine Methode **getTime()** oder **getDate()** oder Ähnliches, mit der Sie experimentieren können.

Ermitteln Sie dann, wie lange ihr Rechner braucht, um 100, 500, 1000, 5000 oder 10000 Zufallszahlen zu sortieren. Wenn Ihr Rechner zu schnell dafür ist, können Sie die Zahl der Elemente noch weiter erhöhen. Wenn Ihr Rechner zu langsam sein sollte, nehmen Sie einfach weniger Zahlen. Falls Sie bei der Recherche nach **getTime()** oder **getDate()** oder vergleichbaren Methoden keinen Erfolg hatten, machen Sie den Array einfach so groß, dass die Sortierzeiten im Sekundenbereich liegen. Sie können dann eine Stoppuhr verwenden, um die Sortierzeiten zu ermitteln.

Stellen Sie Ihre Ergebnisse graphisch dar, indem Sie mit Bleistift und Papier ein entsprechendes Diagramm zeichnen.

Übung 8.3 - 2B (9 Punkte)

Hier die gleiche Aufgabe für Leute, die weder eine Stoppuhr besitzen noch eine eigene Stoppuhr programmieren können:

Ergänzen Sie die Klasse **Liste** um einen internen **Zähler**. Jedes Mal, wenn zwei Zahlen verglichen werden, soll der Zähler um 1 inkrementiert werden. Jedes Mal, wenn zwei Zahlen vertauscht werden müssen, soll der Zähler um 3 inkrementiert werden, denn ein Tauschvorgang besteht ja aus drei Zuweisungen.

Ermitteln Sie dann experimentell die Anzahl der zum vollständigen Sortieren notwendigen Operationen für 100 - 500 - 1000 - 5000 - 10000 Zahlen. Stellen Sie Ihre Ergebnisse graphisch dar, indem Sie auf Papier ein entsprechendes Diagramm zeichnen (6 Punkte).

Wenn Sie es ganz genau machen wollen, müssten Sie auch bei jedem Vergleich in den for-Schleifen den Zähler um 1 hochzählen. Also auch dann, wenn zum Beispiel überprüft wird, ob die Laufvariable bereits den Endwert erreicht hat. Und natürlich müsste auch das Inkrementieren der Laufvariable berücksichtigt werden (weitere 3 Punkte).

Sie machen natürlich nur eine der beiden Übungen 8.3 - 2A bzw. 2B, es handelt sich hier um Alternativ-Aufgaben.

8.4 Der Selectionsort (I2,I4 / A,D)

Auch das **Sortieren durch Auswählen** ist ein Sortierverfahren, in seiner Geschwindigkeit etwas schneller als der Bubblesort. Allerdings ist der Bubblesort einfacher zu programmieren.

Das Verfahren

Am Anfang ist der sortierte Teil **S** des Arrays leer, und der unsortierte Teil **U** enthält *alle* Elemente. In **U** wird jetzt das *kleinste* Element gesucht und mit dem *ersten* Element von **U** vertauscht. Nach dem ersten Sortierschritt hat **U** die Länge N-1 (N = Zahl der Array-Elemente), und **S** die Länge 1.

Nun wird wieder das erste Element von **U** mit dem kleinsten Element von **U** vertauscht. Danach ist **S** wieder um 1 gewachsen, **U** um 1 geschrumpft. So geht das weiter, bis das Ende des Arrays erreicht ist.

Ein Beispiel

Ein Array soll aus sechs int-Zahlen bestehen:

8	5	9	2	1	3
U					

Schritt 1

Wir suchen nun die kleinste Zahl in **U** und vertauschen sie mit der ersten Zahl von **U**:

1	5	9	2	8	3
U					

Der sortierte Teil **S** des Arrays besteht jetzt aus dem ersten Element, der unsortierte Teil **U** ist um ein Element kleiner geworden:

1	5	9	2	8	3
S	U				

Schritt 2

1	5	9	2	8	3
S	U				

Wir suchen wieder die kleinste Zahl in **U** und vertauschen sie mit der ersten Zahl von **U**:

1	2	9	5	8	3
S	U				

Damit ist **S** wieder um 1 gewachsen, **U** um 1 geschrumpft:

1	2	9	5	8	3
S		U			

Weitere Schritte

Die nächsten Schritte sollen im Schnelldurchgang dargestellt werden; das Verfahren an sich sollte jetzt klar sein.

Die Situation nach dem 3. Sortierschritt:

1	2	3	5	8	9

Der Array ist jetzt schon sortiert; allerdings erkennt der Algorithmus dies noch nicht. Daher wird ein vierter Sortierschritt angeschlossen, bei dem aber nichts Wesentliches passiert, da die 5 gleichzeitig die kleinste Zahl wie auch die erste Zahl von **U** ist.

1	2	3	5	8	9

Der Sortieralgorithmus weiß immer noch nicht, ob der Array schon fertig sortiert ist, daher folgt ein fünfter Sortierschritt:

1	2	3	5	8	9

Da **U** nur noch aus *einem* Element besteht, *muss* der Array sortiert sein und der Algorithmus kann terminieren.

Implementationshinweise

Für die Implementierung des Selectionsort-Algorithmus ist nur *eine* for-Schleife notwendig.

In jedem Schleifendurchgang wird die erste Zahl des unsortierten Teils mit der kleinsten Zahl des unsortierten Teils vertauscht. Dazu können wir wieder die Methode **tausche()** der Klasse **Liste** einsetzen.

Die Hauptschwierigkeit beim Selectionsort ist es, eine Methode zu schreiben, die die *Position* der kleinsten Zahl im *unsortierten* Teil des Arrays liefert. Es darf also nicht der *gesamte* Array **S+U** durchsucht werden, sondern nur der *unsortierte* Teil **U**.

Übung 8.4 - 1 (3 Punkte)

Erweitern Sie die Klasse **Liste** um eine Methode, welche die Zahlen mithilfe des Selectionsort-Algorithmus sortiert.

Übung 8.4 - 2 (3 oder 6 Punkte)

Vergleichen Sie die Laufzeit des Bubblesort mit der des Selectionsort. Sie können dazu recherchieren und anschließend vortragen (3 Punkte) oder ein entsprechendes Testprogramm schreiben, das die beiden Algorithmen vergleicht (6 Punkte).

8.5 Insertionsort (I2,I4 / A,D)

Der Insertionsort begegnet uns - ohne dass wir normalerweise Notiz davon nehmen - beim Kartenspielen. Beim Skatspiel zum Beispiel nimmt man die ersten drei Karten auf und ordnet sie sortiert in der Hand an. Dann nimmt man die vierte Karte und sortiert sie *gleich an der richtigen Stelle* ein. Jetzt kommt die fünfte Karte, auch sie wird *gleich an der richtigen Stelle* eingeordnet und so weiter. Damit haben wir das Grundprinzip des **Sortierens durch direktes Einfügen** kennengelernt, wie der **Insertionsort** auch genannt wird.

Das Verfahren

Zu Beginn besteht der sortierte Teil **S** aus dem ersten Array-Element, der unsortierte Teil **U** enthält alle anderen Elemente.

Der Algorithmus funktioniert so: Das **erste** Element von U wird gewählt und an die **richtige Stelle** in den bereits sortierten Teil **S** eingefügt. Dadurch wächst **S** um 1, und **U** wird um 1 kleiner.

Der zweite Schritt und die folgenden Schritte funktionieren ganz genau so: Es wird immer das erste Element von **U** korrekt in **S** eingefügt. Das machen wir uns wieder an einem Beispiel klar.

Ein Beispiel

8	5	9	2	1	3
U					

erster Sortierschritt

S besteht aus dem ersten Element des Arrays, **U** aus dem Rest des Arrays:

8	5	9	2	1	3
S	U				

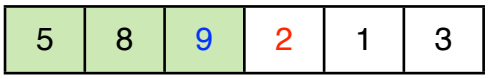
Im ersten Sortierschritt wird nun die *erste Zahl* von **U** genommen und in **S** eingefügt - und zwar an der richtigen Position. Die erste Zahl von **U** ist die 5. Da die 5 *kleiner* ist als die 8, muss sie *vor* der 8 eingefügt werden:

5	8	9	2	1	3
S		U			

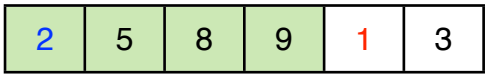
Der sortierte Teil ist jetzt um 1 gewachsen.

nachfolgende Sortierschritte

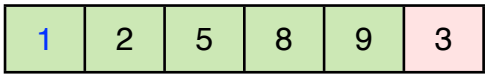
Im Grunde geht es genau so weiter wie beim ersten Sortierschritt beschrieben. Als nächstes muss die 9 in den bereits sortierten Teil eingefügt werden. Da die 9 größer ist als die größte Zahl von **S** (8), passiert gar nichts. Lediglich die Grenze zwischen **S** und **U** wird verschoben:



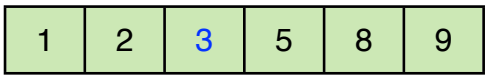
Nun wird die 2 in **S** eingefügt. Da sie kleiner ist als die erste Zahl des sortierten Teils, wird die 2 noch vor der 5 eingefügt:



Ähnliches passiert mit der 1:

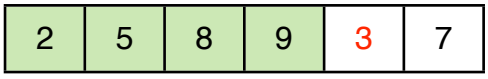


Und mit der 3, welche zwischen der 2 und der 5 eingefügt werden muss.:

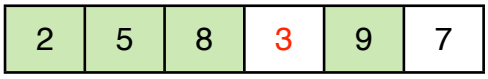


Eine mögliche Implementation

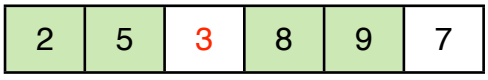
Betrachten wie dazu folgendes Beispiel; der sortierte Teil besteht bereits aus vier Elementen:



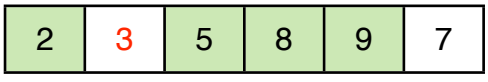
Die 3 soll in **S** eingefügt werden. Sie wird zunächst mit dem *letzten* Element von **S** verglichen. Nun ist die 3 kleiner als die 9, muss also auf jeden Fall links von der 9 eingefügt werden. Daher tauschen wir jetzt die 3 mit der 9, Vergleich und Tausch kosten zusammen vier Rechenoperationen:



Nun müssen wir die 3 mit der 8 vergleichen und tauschen, was wieder vier Operationen kostet:



Auch der nächste Schritt kostet vier Rechenoperationen:

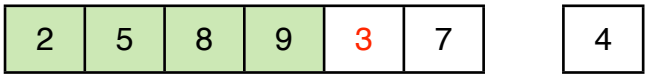


Wir sind jetzt also bei 12 Rechenoperationen angekommen. Ein letzter Vergleich führt zu dem Schluss, dass die Einfügeposition gefunden wurde, denn die 3 ist *nicht kleiner* als die 2. Damit ist die 3 korrekt einsortiert, außerdem ist der sortierte Teil durch dieses reihenweise Tauschen um Eins ge-

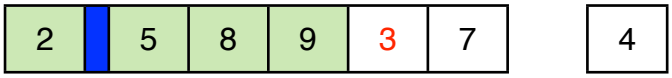
wachsen. 13 Rechenoperationen hat das Einfügen der 3 gekostet, das ist relativ aufwändig! Es gibt aber einen besseren Algorithmus, und den werden wir jetzt kennenlernen.

Ein besserer Insertionsort-Algorithmus

Wir speichern den *Index* der einzusortierenden Zahl 3 in einer Hilfsvariablen (Kästchen ganz rechts) - das ist *eine* Rechenoperation:



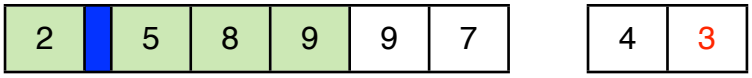
Anschließend muss die Einfügestelle für die 3 gesucht werden. Nach vier Vergleichen (= vier Rechenoperationen) ist die Einfügestelle für die 3 gefunden, nämlich zwischen 2 und 5.



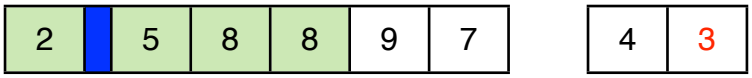
Leider kann man in einen Array nicht einfach so irgendwo eine Zahl einfügen, das geht nur bei dynamischen Datenstrukturen.

Zunächst wird der *Wert* der 3 in einer zweiten Hilfsvariablen gesichert:

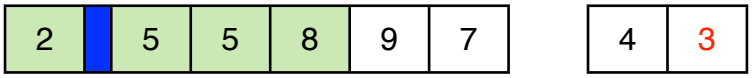
Nun kann die "alte" 3 durch die 9 überschrieben werden, was nur eine Rechenoperation kostet:



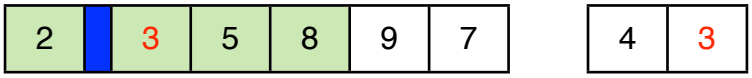
Was wird jetzt gemacht? Die "alte" 9 wird mit der 8 überschrieben:



und dann wird die "alte" 8 mit der 5 überschrieben:



Die 5 ist jetzt zweimal vorhanden, und die "alte" 5 kann durch die 3 überschrieben werden, die in der zweiten Hilfsvariable gespeichert ist:



Der Durchgang ist beendet, und der sortierte Teil des Arrays ist um ein Element gewachsen:

Im letzten Bild ist schon angedeutet, dass jetzt die 7 in den sortierten Teil einsortiert werden muss.

Wir haben für das Einsortieren der 3 nach dem hier beschriebenen Algorithmus genau 10 Rechenoperationen benötigt, das ist deutlich weniger als bei der ersten Implementation, die wir uns angesehen hatten.

Übung 8.5 - 1 (3 Punkte)

Erweitern Sie das "alte" Projekt mit der Klasse **Liste** um eine Methode, die die Zahlen mithilfe des besseren Insertionsort-Algorithmus sortiert.

Übung 8.5 - 2 (4 Punkte)

Vergleichen Sie die Laufzeit des Bubblesort mit der des Insertionsort, indem Sie ein entsprechendes Testprogramm schreiben, das die beiden Algorithmen vergleicht (4 Punkte).

Übung 8.5 - 3 (6 Punkte)

Schreiben Sie ein Testprogramm, das alle drei bisher behandelten Sortieralgorithmen vergleicht und die Resultate graphisch darstellt.

Wie Sie das Programm entwickeln, ist Ihnen überlassen. In den Folgen 5 und 6 haben Sie Java-Applets kennengelernt; Sie wissen, wie man einfache Graphiken zeichnet, und Sie kennen sich mit Buttons aus. Also sollten Sie kein Problem haben, ein schönes Test-Applet zu schreiben, in dem das Zeitverhalten der drei Sortieralgorithmen ermittelt und graphisch dargestellt wird.

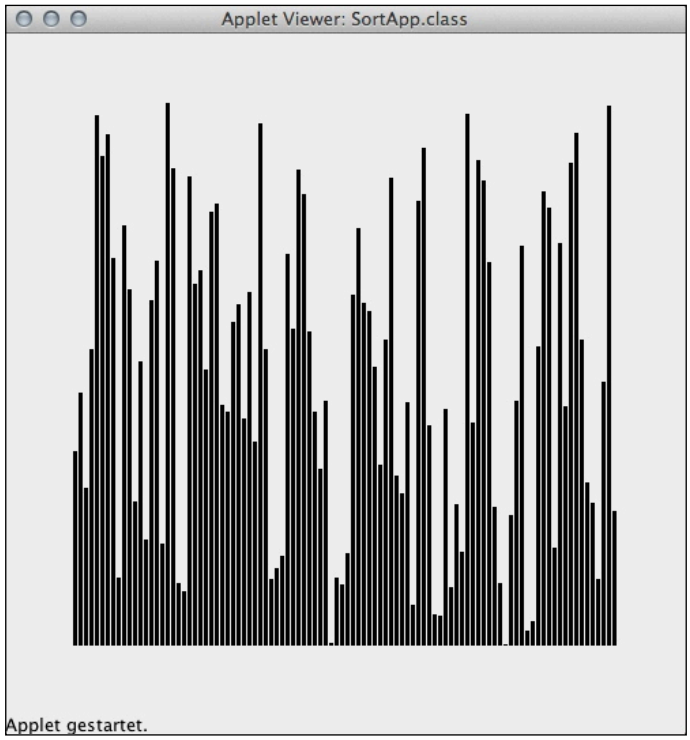
8.6 Visualisierung der Sortieralgorithmen

8.6.1 Problemstellung

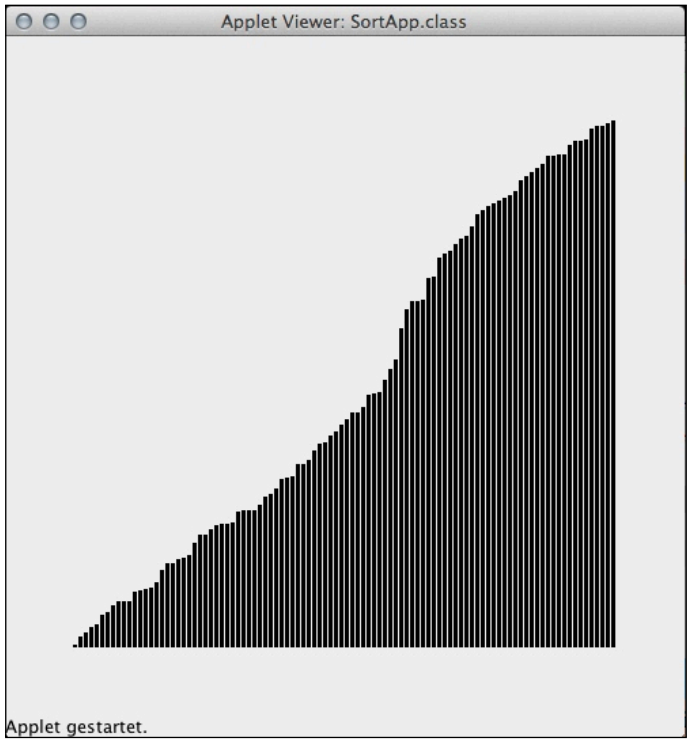
Schauen Sie sich den folgenden Quelltext eines Java-Applets an:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class SortApp extends JApplet implements Runnable
{
    Thread uhr;
    Liste zahlen;

    public void init()
    {
        zahlen = new Liste();
        zahlen.erzeugen();
    }

    public void start()
    {
        if (uhr == null)
        {
            uhr = new Thread(this);
            uhr.start();
        }
    }

    public void run()
    {
        for (int i=0; i<100; i++)
        {
            zahlen.insertionsortNextStep(i);
            repaint();
            try { Thread.sleep(200); }
            catch (InterruptedException e) { }
        }
    }

    public void paint(Graphics g)
    {
        zahlen.paint(g);
    }
}
```

Dieses Applet erzeugt in der **init()**-Methode ein Objekt zahlen der Klasse **Liste**; in der **paint()**-Methode werden die Zahlen dann als **Balkendiagramm** dargestellt.



***8.6-1** Die noch nicht sortierten Zahlen als Balkendiagramm*



***8.6-2** Die Zahlen dem Ausführen eines Sortier-Algorithmus*

8.6.2 Ein Applet mit einem Timer

In der Methode

```
public void start()
 {
     if (uhr == null)
     {
         uhr = new Thread(this);
         uhr.start();
     }
 }
```

des Applets wird das **Timer**-Objekt **uhr** initialisiert und gestartet, falls das noch nicht der Fall war.

Die **run()**-Methode ist die wichtigste Methode eines Timer-Applets:

```
public void run()
 {
     for (int i=0; i<100; i++)
     {
         zahlen.insertionsortNextStep(i);
         repaint();
         try { Thread.sleep(200); }
         catch (InterruptedException e) { }
     }
 }
```

Die for-Schleife ruft 100 Durchgänge einer Insertionsort-Methode auf; zur Steuerung dient die Laufvariable `i`. Nach jedem Durchgang wird **repaint()** aufgerufen, so dass die Zahlen neu gezeichnet werden können. Auf diese Weise kann man den Fortschritt des Sortierens beobachten.

Dann wird versucht, den Prozess für 200 Millisekunden anzuhalten, damit der Benutzer des Programms überhaupt etwas sehen kann. Andernfalls würden die Zahlen so schnell sortiert, dass man gleich das Balkendiagramm mit den fertig sortierten Zahlen sehen würde. Wir wollen den Sortieralgorithmus aber bei seiner Arbeit beobachten, also müssen wir den Vorgang verlangsamen. Dazu dient die 200 ms-Pause.

Falls aus irgendwelchen Gründen dieser Versuch (`try`) misslingt, tritt der `catch`-Fall ein. Es wird dann eine geordnete Fehlermeldung ausgegeben.

So viel zum Thema Timer. Im Abschnitt 9.3 benötigen wir noch mal einen Timer, wenn wir mit Hilfe eines zweidimensionalen Arrays einen zellulären Automaten programmieren.

8.6.3 Übungen

Beginnen wir mit der Haupt-Übung, die am meisten Punkte bringt:

Übung 8.6 - 1 (8 Punkte)

Ergänzen Sie die Klasse **Liste** um die noch fehlende Methode

public void insertionSortNextStep(int i)

welche jeweils *einen* Durchgang des Insertionsort ausführt. Nach dem ersten Aufruf der Methode umfasst der sortierte Teil also genau ein Element. Nach dem zweiten Aufruf durch die **run()**-Methode sind bereits zwei Zahlen sortiert und so weiter. Nach dem letzten Aufruf sind dann alle Zahlen sortiert.

Nicht alle Schüler werden mit dieser Übung auf Anhieb erfolgreich sein, sie ist doch recht schwer. Daher gibt es alternativ ein paar einfachere Übungen, von denen Sie mit Sicherheit eine schaffen werden.

Übung 8.6 - 1A (3 Punkte)

Ergänzen Sie die Klasse **Liste** um die noch fehlende Methode

public void insertionSort()

welche den Insertionsort ausführt. Verwenden Sie dann das folgende Java-Applet, um den Insertionsort zu testen.

```
import java.awt.*;
import javax.swing.*;
public class SortApp extends JApplet
{
    Liste zahlen;

    public void init()
    {
       zahlen = new Liste();
       zahlen.erzeugen();
       zahlen.insertionsort();
    }

    public void paint(Graphics g)
    {
        zahlen.paint(g);
    }
}
```

Wenn Ihnen diese Variante gelungen ist, können Sie sich ja doch noch einmal an der Hauptübung probieren; den Quelltext des Applets haben Sie ja zur Verfügung, sie müssen nur noch den Insertionsort zu umprogrammieren, dass das Applet jeweils nur einen Durchgang ausführt.

Wenn Sie mit dem Timer keine Probleme haben, dafür aber mit der Umgestaltung des Insertionsort, dann ersetzen Sie doch einfach den Insertionsort durch den einfacher zu programmierenden Selectionsort.

Übung 8.6 - 1B (6 Punkte)

Wie Hauptübung 8.6 - 1, allerdings mit Selectionsort statt Insertionsort.

8.6.4 Ergänzungsübungen

Natürlich kann man das Applet noch schöner machen, als es hier der Fall ist.

Übung 8.6 - 2 (3 Punkte)

Führen Sie die Hauptübung oder die Alternativübung 8.6 - 1B durch.

Verändern Sie dann die **paint()**-Methode der Klasse **Liste** so, dass der bereits sortierte Teil des Arrays in einer anderen Farbe dargestellt wird.

Übung 8.6 - 3 (5 Punkte)

Schauen Sie sich in Folge 6 noch einmal an, wie man mit Buttons in einem Java-Applet umgeht. Erweitern Sie das Applet dann so, dass man mit Hilfe von drei Buttons zwischen den drei Sortieralgorithmen Bubblesort, Selectionsort und Insertionsort wählen kann.

Übung 8.6 - 4 (3 Punkte)

Erweitern Sie das Programm aus Übung 8.6 - 3 so, dass die zum Sortieren benötigten Befehle (Vergleiche, Zuweisungen etc.) mitgezählt und angezeigt werden.

Folge 9 - Zweidimensionale Arrays

Unterrichtsvorhaben 1

9.1 Eine Notenliste für eine Klasse

Ein Lehrer möchte die Zensuren der Schüler seiner Klasse übersichtlich speichern. Jeder Schüler hat 10 Fächer, und die Klasse besteht aus 30 männlichen und weiblichen Schülern. Die Noten eines Schülers kann man ganz leicht in einem Array speichern, der 10 Elemente umfasst:

```
import java.util.Random;

public class SNoten
{
    int[] noten;
    Random wuerfel;

    public SNoten()
    {
       noten = new int[10];
       wuerfel = new Random();
    }

    public void erzeugen()
    {
       for (int i=0; i<10; i++)
          noten[i] = wuerfel.nextInt(6)+1;
    }

    public void ausgeben()
    {
       for (int i=0; i<9; i++)
          System.out.print(noten[i] + " / ");
       System.out.println(noten[9]);
    }
}
```

Dieses Programm zeigt eine einfache Version der Notenverwaltung, die dem Lehrer zudem die mühsame Arbeit der Notenfindung erleichtert.

Das Programm verwaltet aber nur die Noten eines einzelnen Schülers, daher heißt die Klasse auch SNoten. Wir wollen aber die Noten einer ganzen Klasse verwalten.

Schritt 2 - Eine Notenliste für die ganze Klasse

```
import java.util.Random;

public class KNoten
{
    int[][] noten;
    Random wuerfel;

    public KNoten()
    {
       noten = new int[30][10];
       wuerfel = new Random();
    }

    public void erzeugen()
    {
       for (int s=0; s<30; s++)
          for (int i=0; i<10; i++)
             noten[s][i] = wuerfel.nextInt(6)+1;
    }

    public void ausgeben()
    {
       for (int s=0; s<30; s++)
       {
          System.out.print("Noten fuer Schueler "+(s+1)+":\t");
          for (int i=0; i<9; i++)
             System.out.print(noten[s][i] + " / ");
          System.out.println(noten[s][9]);
       }
    }
}
```

Die Ausgabe des Programms sehen Sie auf der nächsten Seite.

Neu an der Klasse **KNoten** ist die Verwendung eines zweidimensionalen Arrays. Einen zweidimensionalen Array müssen Sie sich so ähnlich vorstellen wie ein Schachbrett. Ein Schachbrett besteht aus acht Reihen mit jeweils acht Feldern. Der in der Klasse **KNoten** verwendete zweidimensionale Array besteht aus 30 Reihen mit jeweils zehn Feldern.

Deklaration eines zweidimensionalen Arrays

Syntax der Array-Initialisierung

```
datentyp[][] bezeichner;
```

Beispiele:

```
int[][] noten;
double[][] einnahmen;
Gegenstand[][] schrank;
```

Initialisierung eines zweidimensionalen Arrays

Syntax der Array-Initialisierung

```
bezeichner = new Datentyp[AnzahlReihen][AnzahlFelder];
```

Beispiele:

```
noten = new int[30][10];
einnahmen = new double[12][20];
schrank = new Gegenstand[4][12];
```

Das letzte Beispiel zeigt, dass man auch von Objekten einer Klasse zweidimensionale Array anlegen kann. Der Schrank enthält vier Schubladen, und in jede Schublade passen maximal zwölf Gegenstände.

Hier nun die Ausgabe der Klasse **KNoten**:

BlueJ: Konsole – Notenliste_neu V2

```
Noten fuer Schueler 1:  5 / 1 / 3 / 5 / 4 / 6 / 6 / 4 / 5 / 1
Noten fuer Schueler 2:  3 / 3 / 4 / 4 / 6 / 5 / 2 / 5 / 4 / 4
Noten fuer Schueler 3:  4 / 3 / 2 / 2 / 6 / 1 / 6 / 5 / 5 / 2
Noten fuer Schueler 4:  1 / 2 / 4 / 4 / 2 / 3 / 2 / 1 / 1 / 5
Noten fuer Schueler 5:  6 / 1 / 4 / 4 / 5 / 3 / 5 / 3 / 4 / 1
Noten fuer Schueler 6:  2 / 6 / 3 / 5 / 6 / 6 / 5 / 1 / 3 / 5
Noten fuer Schueler 7:  2 / 3 / 6 / 1 / 4 / 2 / 1 / 2 / 3 / 2
Noten fuer Schueler 8:  4 / 1 / 3 / 3 / 6 / 3 / 1 / 1 / 2 / 4
Noten fuer Schueler 9:  6 / 1 / 6 / 2 / 3 / 5 / 1 / 2 / 2 / 6
Noten fuer Schueler 10: 3 / 1 / 2 / 4 / 5 / 1 / 1 / 4 / 3 / 6
Noten fuer Schueler 11: 1 / 1 / 1 / 5 / 4 / 1 / 5 / 6 / 4 / 6
Noten fuer Schueler 12: 2 / 5 / 4 / 3 / 3 / 4 / 5 / 1 / 1 / 5
Noten fuer Schueler 13: 3 / 6 / 5 / 2 / 1 / 1 / 3 / 6 / 5 / 3
Noten fuer Schueler 14: 2 / 2 / 6 / 2 / 4 / 6 / 4 / 4 / 6 / 1
Noten fuer Schueler 15: 6 / 2 / 3 / 4 / 6 / 6 / 6 / 1 / 5 / 3
Noten fuer Schueler 16: 4 / 5 / 4 / 3 / 6 / 2 / 3 / 6 / 2 / 1
Noten fuer Schueler 17: 3 / 3 / 6 / 4 / 5 / 2 / 1 / 1 / 4 / 5
Noten fuer Schueler 18: 2 / 4 / 5 / 2 / 6 / 6 / 3 / 4 / 1 / 2
Noten fuer Schueler 19: 5 / 6 / 4 / 1 / 5 / 1 / 5 / 3 / 1 / 3
Noten fuer Schueler 20: 3 / 6 / 6 / 3 / 2 / 3 / 5 / 1 / 2 / 3
Noten fuer Schueler 21: 6 / 4 / 4 / 3 / 1 / 2 / 3 / 4 / 6 / 6
Noten fuer Schueler 22: 1 / 3 / 2 / 6 / 5 / 1 / 4 / 2 / 3 / 1
Noten fuer Schueler 23: 1 / 3 / 4 / 2 / 3 / 2 / 2 / 6 / 4 / 5
Noten fuer Schueler 24: 6 / 4 / 5 / 2 / 3 / 2 / 6 / 6 / 3 / 6
Noten fuer Schueler 25: 6 / 3 / 2 / 2 / 1 / 4 / 2 / 3 / 4 / 4
Noten fuer Schueler 26: 2 / 2 / 6 / 3 / 6 / 2 / 2 / 6 / 4 / 6
Noten fuer Schueler 27: 1 / 3 / 5 / 1 / 5 / 4 / 4 / 3 / 1 / 4
Noten fuer Schueler 28: 2 / 1 / 5 / 2 / 1 / 3 / 6 / 4 / 1 / 5
Noten fuer Schueler 29: 3 / 4 / 1 / 3 / 2 / 5 / 3 / 3 / 1 / 4
Noten fuer Schueler 30: 6 / 6 / 2 / 1 / 6 / 4 / 4 / 6 / 2 / 2
```

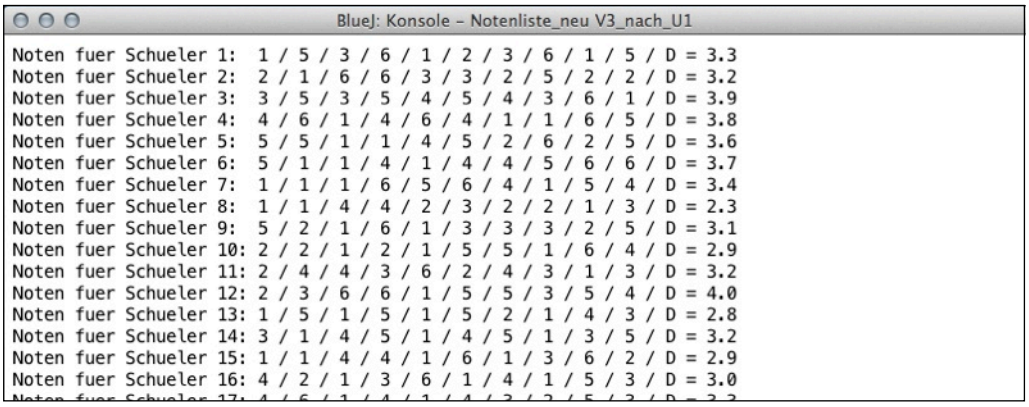
***9.1 - 1** Eine Ausgabe der Klasse KNoten*

Schritt 3 - Übungen

Übung 9.1 - 1 (3 Punkte)

Erweitern Sie die **ausgeben()**-Methode der Klasse **KNote** so, dass am Ende einer jeden Reihe die Durchschnittsnote der Schüler angezeigt wird.

Hier ein Ausgabebeispiel (Ausschnitt):



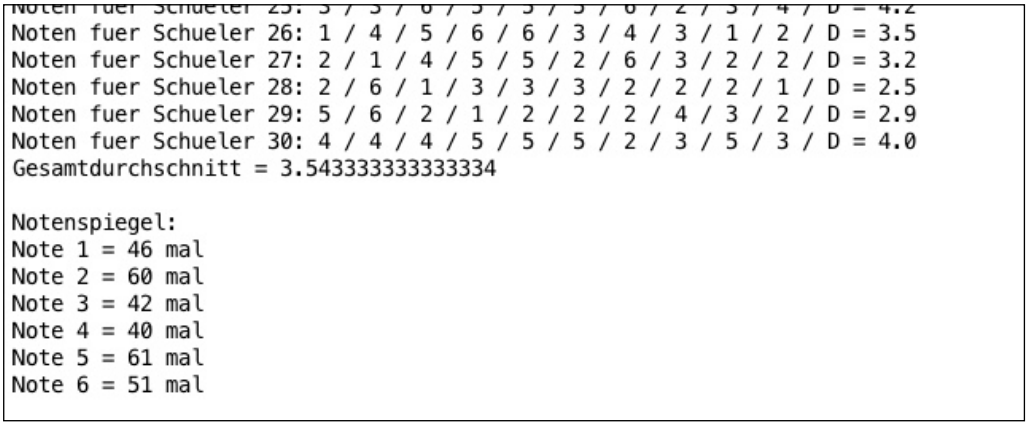
***9.1 - 2** Ein Ausgabebeispiel für Übung 9.1-1*

Übung 9.1 - 2 (2 Punkte)

Erweitern Sie die **ausgeben()**-Methode der Klasse **KNote** so, dass in der letzten Zeile der Ausgabe die Durchschnittsnote der Klasse angezeigt wird.

Übung 9.1 - 3 (3 Punkte)

Ergänzen Sie die Klasse **KNote** um eine Methode **zeigeNotenspiegel()**, die einen Notenspiegel ausgibt, also die Anzahl der Einsen, Zweien, Dreien und so weiter in der Klasse.



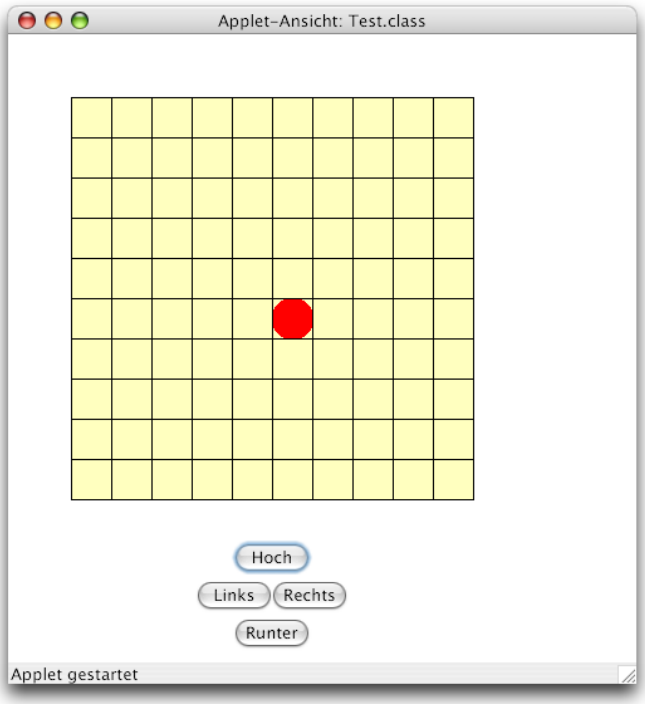
***9.1-3** Die Ausgabe des Notenspiegels könnte so aussehen*

Unterrichtsvorhaben 2

9.2 Eine Graphikanwendung (fakultativ)

Dieser Workshop dient allein der Vertiefung der Graphikprogrammierung sowie des Umgangs mit zweidimensionalen Arrays. Übungen mit Punkten sind für diesen fakultativen Workshop nicht vorgesehen.

Betrachten Sie folgenden Screenshot:



***9.2 - 1** So sieht die fertige Graphikanwendung aus*

Die Graphikanwendung, die Sie in diesem Unterrichtsvorhaben programmieren soll, ist nicht besonders kompliziert, könnte aber das Grundgerüst für ein eigenes kleines Spiel werden.

Sie sehen eine Spielfläche, die aus 10 x 10 Spielfeldern besteht. Ein roter Spielstein kann nun mithilfe der vier Buttons gesteuert werden. Das war's eigentlich schon.

Die Spielfläche besteht aus 10 x 10 Spielfeldern. Wir werden in dem Workshop so vorgehen, dass wir zuerst mal eine Klasse **Feld** programmieren und dann einen zweidimensionalen Array dieser Spielfelder erzeugen.

Aber eines nach dem anderen...

Schritt 1 - Klasse Feld

Wir verfahren nach dem bewährten Prinzip der **Bottom-Up-Programmierung** und entwickeln zunächst eine Klasse für ein einzelnes Spielfeld (eines der kleinen Quadrate im Applet).

Ein Feld soll quadratisch dargestellt werden, und ein Attribut **status** speichert, ob es sich um ein *leeres* Feld oder ein von der Figur *besetztes* Feld handelt. Außerdem ist es notwendig, die relativen Koordinaten des Feldes mit zu speichern, wie wir später noch sehen werden. Hier ist der Quelltext der Klasse **Feld**, Version 1:

```
import java.awt.*;

public class Feld
{   int status;
    int x,y;

  public Feld(int x, int y)
  {
      status = 0;
      this.x = x;
      this.y = y;
  }

public void anzeigen(Graphics g)
  {
     if (status == 0)
     {
        g.setColor(new Color(255,255,191)); // hellgelb
        g.fillRect(50+x*40,50+y*40,40,40);
        g.setColor(new Color(0,0,0)); // schwarz
        g.drawRect(50+x*40,50+y*40,40,40);
     }
  }
}
```

Da können wir noch nichts Besonderes entdecken, nicht wahr? Die wichtigen Informationen werden in primitiven Attributen gespeichert, der Konstruktor initialisiert diese Attribute mit Werten, und die Methode **anzeigen()** stellt das Feld in einem Applet dar.

Schritt 2 - Test-Applet

Jetzt schreiben wir uns ein kleines Test-Applet, welches die Methoden von **Feld** testet. Hier der Quelltext. Wie üblich sind in diesem Skript die Quelltexte von Applets rosa unterlegt, die Quelltexte von normalen Klassen blau.

```
import java.awt.*;
import javax.swing.*;

public class Test extends JApplet
{
    Feld[][] testfeld;

    public void init()
    {
      testfeld = new Feld[10][10];
      for (int x=0; x<10; x++)
        for (int y=0; y<10; y++)
          testfeld[x][y] = new Feld(x,y);
  }

    public void paint(Graphics g)
    {
      for (int x=0; x<10; x++)
        for (int y=0; y<10; y++)
            testfeld[x][y].anzeigen(g);
    }
}
```

Dieses Applet enthält nicht nur ein einzelnes Objekt der Klasse **Feld**, sondern einen zweidimensionalen Array von solchen Feldern.

Schritt 3 - Feld mit Zuständen

Wir ergänzen die **anzeigen()**-Methode der Klasse **Feld** so, dass *zwei* verschiedene Zustände angezeigt werden können:

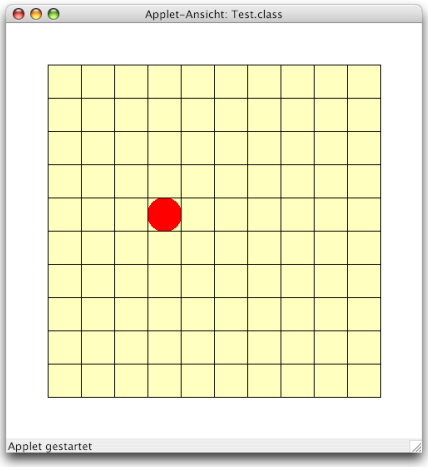
```
    public void anzeigen(Graphics g)
    {
       if (status == 0)
       {
          g.setColor(new Color(255,255,191)); // hellgelb
          g.fillRect(50+x*40,50+y*40,40,40);
          g.setColor(new Color(0,0,0)); // schwarz
          g.drawRect(50+x*40,50+y*40,40,40);
       }

       if (status == 1)
       {
          g.setColor(new Color(255,255,191)); // hellgelb
          g.fillRect(50+x*40,50+y*40,40,40);
          g.setColor(new Color(255,0,0)); // rot
          g.fillOval(50+x*40,50+y*40,40,40);
          g.setColor(new Color(0,0,0)); // schwarz
          g.drawRect(50+x*40,50+y*40,40,40);
       }

    }
```

Falls **status** den Wert 0 hat, wird ein hellgelbes Feld angezeigt, und wenn **status** den Wert 1 hat, so wird ein roter Kreis auf einem hellgelben Feld dargestellt.

In der Klasse **Test** fügen wird am Ende der **init()**-Methode eine Zeile hinzu:

```
       testfeld[3][4].neuerStatus(1);
```

Auf diese Weise soll dann das Feld mit den Koordinaten (3,4) einen roten Kreis enthalten. Wenn wir das Ganze kompilieren und zum Laufen bringen, können wir uns davon überzeugen, dass der bisherige Quelltext funktioniert.



***9.2 - 2** Das Test-Applet nach Schritt 3*

Schritt 4 - Flexible Felder

Wir machen zunächst die Klasse **Feld** flexibler. Die Breite der Felder war bisher auf 40 Pixel festgelegt. Jetzt soll der Benutzer der Klasse selbst entscheiden, wie breit seine Felder sein sollen.

Um dies zu erreichen, müssen wir zwei Dinge erledigen. Erstens muss die Breite des Feldes *innerhalb* der Klasse **Feld** gespeichert werden. Dies sollte kein Problem sein: Ein neues Attribut **breite**, und das Problem ist gelöst. Beim Erzeugen eines Feld-Objektes muss dann die gewünschte Breite als Parameter übergeben werden. Betrachten wir den neuen Konstruktor:

```
public Feld(int x, int y, int b)
{
    status = 0;
    this.x = x;
    this.y = y;
    breite = b;
}
```

Wie sich diese flexiblere Lösung auf das Zeichnen der Felder auswirkt, zeigt der Quelltext der **anzeigen()**-Methode:

```
public void anzeigen(Graphics g)
{
   if (status == 0)
   {
      g.setColor(new Color(255,255,191)); // hellgelb
      g.fillRect(50+x*breite,50+y*breite,breite,breite);
      g.setColor(new Color(0,0,0)); // schwarz
      g.drawRect(50+x*breite,50+y*breite,breite,breite);
   }
   ...
```

Aus Platzgründen wurde hier der zweite Fall `(status == 1)` weggelassen.

Schritt 5 - Applet mit Buttons

Wir wollen jetzt das Applet mit vier **Buttons** ausstatten. Diesmal werden wir keinen Array deklarieren, sondern vier einzelne Buttons, die wir **up**, **down**, **left** und **right** nennen.

```
import java.awt.*;
import javax.swing.*;

public class Test extends JApplet
{
    Feld[][] testfeld;
    Button   up,down,left,right;

    public void init()
    {
      testfeld = new Feld[10][10];
      for (int x=0; x<10; x++)
        for (int y=0; y<10; y++)
          testfeld[x][y] = new Feld(x,y,32);

      testfeld[3][4].neuerStatus(1);

      up    = new Button("Hoch");
      down  = new Button("Runter");
      left  = new Button("Links");
      right = new Button("Rechts");

      add(up);
      add(down);
      add(left);
      add(right);

      setLayout(null);

      up.setBounds   (180,400,60,30);
      down.setBounds (180,460,60,30);
      left.setBounds (150,430,60,30);
      right.setBounds(210,430,60,30);
    }

    public void paint(Graphics g)
    {
      for (int x=0; x<10; x++)
        for (int y=0; y<10; y++)
          testfeld[x][y].anzeigen(g);
    }
}
```

Hier muss nicht viel erläutert werden, Buttons sind Ihnen bereits aus der Folge 6 (Roboter) sowie aus der Folge 8 (Sortieralgorithmen) bekannt. Eine Funktionalität haben die Buttons noch nicht, das kommt im nächsten Schritt.

Schritt 6 - Buttons mit Funktionen

Wir wollen die Buttons nun mit Funktionalität ausstatten, zumindest den up-Button. Wenn auf den up-Button geklickt wird, soll das Feld *über* dem markierten Feld ebenfalls markiert werden. Eine ganz einfache Aufgabe also.

An der Klasse **Feld** müssen wir diesmal nichts ändern. Was für ein Glück!

Die Klasse **Test**, also unser Applet, wird wieder um einen **ActionListener** ergänzt:

```
public class Test extends JApplet implements ActionListener
```

Entsprechendes gilt für die vier Buttons:

```
    up.addActionListener(this);
    down.addActionListener(this);
    left.addActionListener(this);
    right.addActionListener(this);
```

Und natürlich brauchen wir eine Methode **actionPerformed()**:

```
  public void actionPerformed(ActionEvent event)
  {
     if (event.getSource() == up)
        testfeld[3][3].neuerStatus(1);
     if (event.getSource() == down)
        testfeld[3][5].neuerStatus(1);
     if (event.getSource() == left)
        testfeld[2][4].neuerStatus(1);
     if (event.getSource() == right)
        testfeld[4][4].neuerStatus(1);
     repaint();
  }
```

Diese **actionPerformed()**-Methode ist sehr einfach gestrickt. Eine "Graphik-Demo" kann man das wirklich noch nicht nennen. Unser Ziel: Der rote Spielstein soll über das ganze Spielbrett bewegt werden können. Und natürlich soll das Feld, auf dem er vorher gestanden hat, hinterher wieder den Status 0 haben, also als gelbes leeres Feld gezeichnet werden.

Schritt 7 - Die Klasse Spielbrett

Jetzt kommt eine ziemlich gravierende Umstellung des Graphik-Programms. Wir erzeugen nämlich eine neue Klasse **Spielbrett**. Schauen wir uns den Quelltext dieser Klasse an:

```
import java.awt.*;

public class Spielbrett
{
    Feld[][] feld;
    int xpos, ypos;

    public Spielbrett(int xStart, int yStart, int breite)
    {
      xpos = xStart;
      ypos = yStart;
      feld = new Feld[10][10];

      for (int x=0; x<10; x++)
        for (int y=0; y<10; y++)
          feld[x][y] = new Feld(x,y,breite);

      feld[xpos][ypos].neuerStatus(1);
    }

    public void anzeigen(Graphics g)
    {
      for (int x=0; x<10; x++)
        for (int y=0; y<10; y++)
          feld[x][y].anzeigen(g);
    }
}
```

Das ganze Projekt wurde **umstrukturiert**. Alle Anweisungen des Applets, die für die Verwaltung der Felder zuständig waren, sind nun **in eine neue Klasse ausgelagert** worden. Die Klasse **Spielbrett** enthält jetzt den zweidimensionalen Array mit den 100 Feldern, und die Klasse **Spielbrett** ist für die Deklaration, Initialisierung und für das Anzeigen der 100 Felder zuständig.

Refactoring

Eine solche gravierende Umstrukturierung eines Software-Projektes wird auch als **Refactoring** bezeichnet (siehe Wikipedia-Artikel "Refactoring").

Durch diese Umstrukturierung hat sich der Quelltext des Applets stark vereinfacht:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Test extends JApplet implements ActionListener
{
    Button     up,down,left,right;
    Spielbrett brett;

    public void init()
    {
      brett = new Spielbrett(5,5,32);
      ...
    }

    public void actionPerformed(ActionEvent event)
    {
      ...
    }

    public void paint(Graphics g)
    {
      brett.anzeigen(g);
    }

}
```

Keine doppelten for-Schleifen mehr in der **init()**- oder **paint()**-Methode, sondern nur noch ein knapper Aufruf des Spielbrett-Konstruktors bzw. der **Spielbrett**.**anzeigen()**-Methode.

Refactoring

Genau das ist der Sinn des Refactoring: Ein vorhandenes Projekt soll vereinfacht werden, es soll leichter lesbar werden und daher auch leichter gepflegt werden können.

Schritt 8 - Bewegen der Figur

Jetzt wird es langsam kompliziert. Wir müssen genau überlegen: Was soll geschehen, wenn der Benutzer auf den up-Button klickt (oder den down-, left- oder right-Button)?

Die Figur geht ein Feld nach oben - falls dort noch Platz ist. Das neue Feld hat dann den Status 1, während das alte Feld hinterher den Status 0 hat. Durch Aufruf der **anzeigen()**-Methode der Klasse **Spielbrett** werden dann alle Felder neu gezeichnet. Die Verwaltung der einzelnen Felder obliegt der Klasse **Spielbrett**. Also muss sich auch **Spielbrett** um das Bewegen der Figur kümmern. So können wir die **actionPerformed()**-Methode des Applets - nach dem Top-Down-Verfahren - ganz einfach entwerfen:

```
public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == up)    brett.hoch();
    if (event.getSource() == down)  brett.runter();
    if (event.getSource() == left)  brett.links();
    if (event.getSource() == right) brett.rechts();

    repaint();
}
```

Das war ja einfach - dank der Top-Down-Methode. Wie die noch zu schreibenden Methoden (das ist mit Top-Down gemeint) genau arbeiten, darum kümmern wir uns später, nämlich in Schritt 9.

Schritt 9 - Besseres Bewegen der Figur

Was hier zu erledigen ist, wurde gerade eben gesagt. Wir müssen die Methoden **hoch()** bis **rechts()** der Klasse **Spielbrett** implementieren. Schauen wir uns mal den Quelltext der **hoch()**-Methode an:

```
public void hoch()
{
   if (ypos > 0)
   {
      feld[xpos][ypos].neuerStatus(0);
      ypos--;
      feld[xpos][ypos].neuerStatus(1);
   }
}
```

Die Position der roten Spielfigur wird durch die neuen Attribute **xpos** und **ypos** verwaltet. Bei der **hoch()**-Methode muss als erstes überprüft werden, ob die Spielfigur überhaupt noch ein Feld nach oben gehen kann. Wenn die y-Koordinate der Figur nämlich bereits den Wert 0 hat, steht die Figur schon ganz oben und kann nicht mehr hoch bewegt werden.

Angenommen, die y-Koordinate der Figur ist > 0, dann passieren drei Dinge:

Erstens: Das alte Feld, auf dem die Figur gestanden hat, bekommt den Status 0:

```
      feld[xpos][ypos].neuerStatus(0);
```

Zweitens: Die Figur bewegt sich nach oben, indem einfach die y-Koordinate um Eins vermindert wird:

```
      ypos--;
```

Drittens: Das Feld, auf dem die Figur jetzt steht, bekommt den Status 1:

```
      feld[xpos][ypos].neuerStatus(1);
```

Auf ähnliche Weise werden die drei anderen Methoden **runter()**, **links()** und **rechts()** programmiert.

Übung 9.2 - 1

Natürlich ist das noch kein Spiel, was wir eben zusammen programmiert haben, sondern nur das Grundgerüst für ein mögliches Spiel. Ein paar Anregungen für Sie: Die Felder könnten einen dritten Status haben, der z.B. so zu interpretieren ist, dass etwas Essbares auf dem Feld liegt. Die Figur könnte am Anfang recht klein sein, und jedes mal, wenn sie auf ein Feld mit etwas Essbarem trifft, könnte ihr Durchmesser um 2 Pixel wachsen, bis sie ihre maximale Größe erreicht hat. Man könnte Wände in das Spielbrett einziehen, die von der Figur nicht passiert werden können, die Wände wiederum könnten Türen enthalten und so weiter und so fort.

Punkte für die Übung sind hier nicht angegeben; Ihr Lehrer entscheidet selbst, wie viele Punkte er Ihnen gibt, wenn Sie eine oder mehrere dieser Anregungen aufgreifen und in das Spiel einbauen.

Unterrichtsvorhaben 3

9.3 Zelluläre Automaten mit 2D-Arrays (fakultativ)

Zunächst etwas Theorie

Zellulärer Automat

Ein Zellulärer Automat ist ein 4-Tupel Z = (R, N, Q, δ), dabei ist

R = ein Raum (z.B. eine zweidimensionale Matrix)

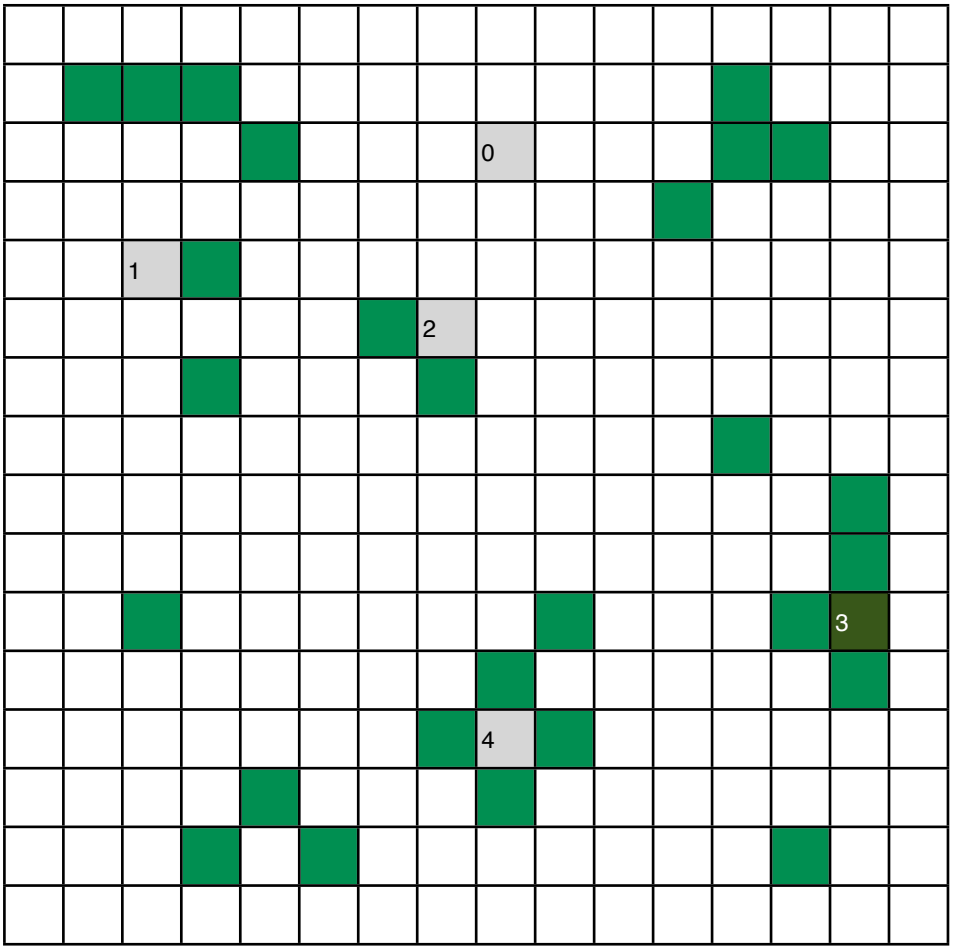
N = eine endliche Nachbarschaft (z.B. die vier Nachbarzellen links, rechts, oben, unten)

Q = eine endliche Zustandsmenge (z.B. { gesund, krank, tot })

δ = eine Überführungs-Funktion

Das hört sich ja sehr theoretisch an, was daran liegen könnte, dass es auch sehr theoretisch ist (immerhin stammt diese Definition aus einem "richtigen" Informatikbuch).

Aus diesem Grund erläutere ich Ihnen die Definition an einem einfachen Beispiel.



***9.3 - 1** Ein Zellulärer Automat mit 256 Zellen, 4er Nachbarschaft und zwei Zuständen*

In der Abbildung sehen wir einen zweidimensionalen zellulären Automaten mit 16 x 16 Zellen, die in je zwei Zuständen vorkommen können, weiß und grün (was man z.B. als "tot" und "lebendig" interpretieren könnte). Jede Zelle hat genau vier Nachbarzellen, nämlich links und rechts sowie oben und unten. Fünf der Zellen wurden besonders gekennzeichnet (grau), die Zahl der "lebenden" Nachbarzellen wurde in diesen Zellen vermerkt.

Damit hätten wir die Komponenten R, N und Q dieses Zellulären Automaten schon einmal näher beschrieben.

R = der abgebildete "Raum" aus 16 x 16 Zellen

N = die Zellen, die sich links, rechts, oberhalb und unterhalb der betrachteten Zelle befinden.

Q = {lebendig, tot}

Es bleibt nur noch die Übergangs-Funktion **δ**. Wir wollen eine solche Funktion **δ** erst einmal anschaulich als Tabelle und anschließend formal definieren.

Die Übergangs-Funktion

Am anschaulichsten stellt man die Übergangsfunktion als Tabelle dar:

Zahl der lebenden Nachbarn	Neuer Zustand
0	0
1	1
2	1
3	0
4	0

***9.3 - 2** Eine einfache Übergangsfunktion*

Wenn eine Zelle *keine* lebenden Nachbarn hat, soll sie im nächsten Zyklus tot sein. Hat eine Zelle *einen oder zwei* lebende Nachbarn, so soll sie im nächsten Zyklus lebendig sein. Bei drei oder vier lebenden Nachbarn soll eine Zelle im nächsten Zyklus tot sein.

Bei dieser Übergangsfunktion hängt der Zustand im nächsten Zyklus nur von der Zahl der lebenden Zellen im aktuellen Zyklus ab, nicht aber vom Zustand der Zelle selbst. Das wäre dann ebenfalls eine Übergangsfunktion, aber eine kompliziertere. Man könnte diese kompliziertere Übergangsfunktion mit einer dreispaltigen Tabelle definieren, wobei die neue Spalte den aktuellen Zustand darstellt.

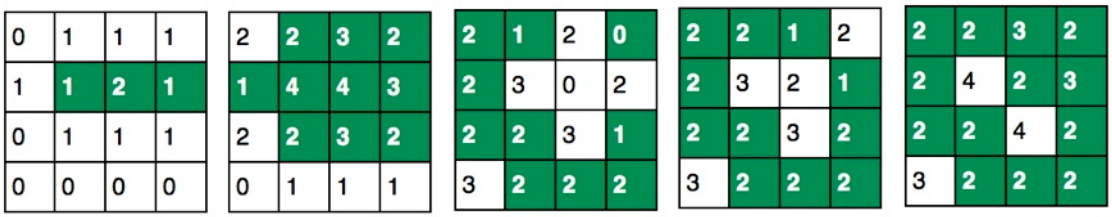
Schauen wir uns nun an, welche Auswirkungen diese Übergangs-Funktion auf die Verteilung der Zellen im Raum hat.

Betrachten wir zunächst die mit der Ziffer "0" markierte Zelle in der Abbildung 9.3 - 1. Diese tote Zelle hat keine lebenden Nachbarn, also wird ihr Zustand in der nächsten Generation wieder "tot" sein. Die mit der Ziffer "1" gekennzeichnete Zelle wird in der nächsten Generation den Zustand 1 oder "lebendig" haben, ebenso die mit der Ziffer "2" gekennzeichnete Zelle. Die lebende Zelle mit

der Ziffer "3" wird sterben, und die tote Zelle "4" wird wegen ihrer vier lebenden Nachbarn in dem Zustand "tot" verbleiben.

Berechnung der nächsten Generationen

Der Zelluläre Automat berechnet jetzt für jede einzelne Zelle mithilfe der Übergangs-Funktion den Folgezustand in der nächsten Generation, realisiert diesen neuen Zustand aber erst dann, wenn er mit der letzten Zelle fertig ist. Somit hängt der Zustand einer Zelle ausschließlich vom aktuellen Zustand der Nachbarzellen ab, und nicht vom bereits errechneten Folgezustand. Schauen wir uns das mal für ein sehr kleines Beispiel an. Ich hoffe, ich habe bei dem manuellen Berechnen der Folgezustände keine Fehler eingebaut:



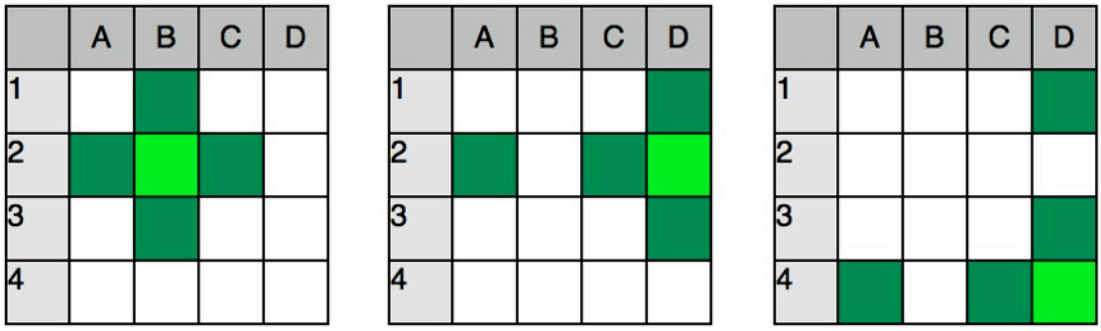
9.3 - 3 *Ein Raum aus 4 x 4 Zellen in fünf Zyklen*

Eine Torus-Welt

Bei der Berechnung der Nachbarschaft sind übrigens die Randzellen sowie die Eckzellen problematisch. Zwangsläufig haben Randzellen nur drei Nachbarn, während Eckzellen sogar nur zwei Nachbarn haben. Entweder muss dies bei der Übergangs-Funktion berücksichtigt werden, was diese natürlich enorm verkompliziert, oder es muss in die geometrische Trickkiste gegriffen werden. Wir verkleben einfach den linken Rand unserer Welt mit dem rechten Rand, so dass wir einen Zylinder erhalten. Und jetzt wird es kompliziert: Wir verkleben den oberen Rand des Zylinders mit dem unteren Rand. So erhalten wir einen sogenannten Torus:

9.3 - 4 *Ein Torus*

Einen Torus kennen Sie sicherlich aus dem Alltag: Ein Fahrradschlauch oder ein Reifen ist im Grunde auch ein Torus. In unserer Torus-Welt hat jetzt jedes Feld vier Nachbarn. Randfelder und Eckfelder gibt es nicht mehr.



9.3 - 5 *Nachbarschaften in einer Torus-Welt*

Wir wollen jetzt einen solchen Zellulären Automaten mithilfe eines Java-Applets simulieren. Dabei wollen wir eine Matrix aus 50 x 50 Zellen mit zwei Zuständen zu einer Torus-Welt zusammenfügen, die Zellen mit dem Zufallsgenerator initialisieren und dann mithilfe eines Timers die Folgegenerationen automatisch erzeugen und zeichnen lassen.

Schritt 1 - Ein Spielfeld

Wir beginnen mit zwei einfachen Java-Klassen, dem eigentlichen Spielfeld und dem Applet zum Anzeigen des Spielfeldes (und später der Buttons und der anderen Bedienelemente). Hier der komplette Quelltext der Klasse **Spielfeld**:

```
import java.util.Random;
import java.awt.*;

public class Spielfeld
{
  int[][] feld;
  Random rand = new Random();

    public Spielfeld()
    {
     feld = new int [50][50];
     for (int y = 0; y < 50; y++)
        for (int x = 0; x < 50; x++)
        {
           int zufall = rand.nextInt(10);
           if (zufall < 9)
                feld[x][y] = 0;
             else
                feld[x][y] = 1;
        }
    }

    public void anzeigen(Graphics g)
    {
       for (int y = 0; y < 50; y++)
        for (int x = 0; x < 50; x++)
        {
           if (feld[x][y] == 1)
              g.setColor(new Color(0,127,0));
           else if (feld[x][y] == 0)
              g.setColor(Color.YELLOW);

           g.fillRect(50+x*8, 20+y*8,8,8);
           g.setColor(new Color(127,127,127));
           g.drawRect(50+x*8, 20+y*8,8,8);
        }
    }
}
```

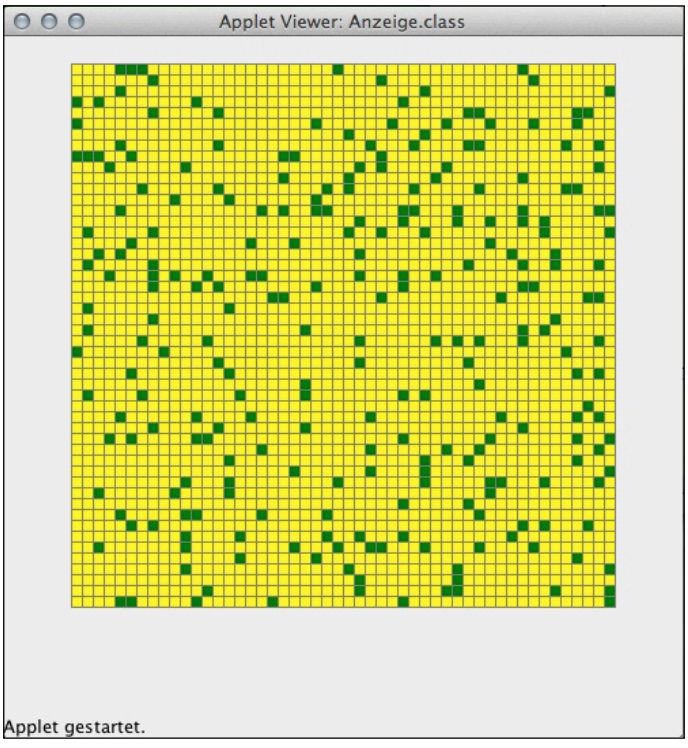
Hier werden die lebendigen Felder durch die Farbe Dunkelgrün und die toten Felder durch die Farbe Gelb repräsentiert. Es steht Ihnen frei, dies zu ändern.

Schritt 2 - Das Applet

```
import java.awt.*;
import javax.swing.*;

public class Anzeige extends JApplet
{
  Spielfeld welt;

  public void init()
  {
     welt = new Spielfeld();
  }

  public void paint(Graphics g)
  {
     welt.anzeigen(g);
  }
}
```

Der Quelltext des Java-Applets ist recht kurz, aber so wollten wir das ja immer halten: Nur das Notwendigste soll in dem Applet-Quelltext stehen, alles andere soll in die untergeordneten Klassen ausgelagert werden.

Und hier die Ausgabe, die uns das Applet liefert:



***9.3 - 6** Die erzeugte Welt*

Schritt 3 - Berechnung der neuen Generation

Wir wollen jetzt eine neue wichtige Methode in die Klasse **Spielfeld** einbauen:

```
public void neuerZyklus()
{
// temporäres Feld berechnen
   for (int y = 0; y < 50; y++)
      for (int x = 0; x < 50; x++)
      {
         int s = nachbarSumme(x,y);
         if (s < 2)
              temp[x][y] = 0;
         else if (s == 2)
              temp[x][y] = 1;
         else if (s > 2)
              temp[x][y] = 0;
      }

// tatsächliches Feld aktualisieren
   for (int y = 0; y < 50; y++)
      for (int x = 0; x < 50; x++)
         feld[x][y] = temp[x][y];
}
```

Sie sehen hier den Quelltext der Methode **neuerZyklus()**. Die beiden geschachtelten for-Schleifen am Anfang der Methode untersuchen jedes der 50 x 50 Felder und ermitteln mithilfe einer noch zu entwickelnden sondierenden Methode **nachbarSumme()** die Zahl der lebenden Nachbarn für jedes Feld. Dabei wird von einer Torus-Welt ausgegangen.

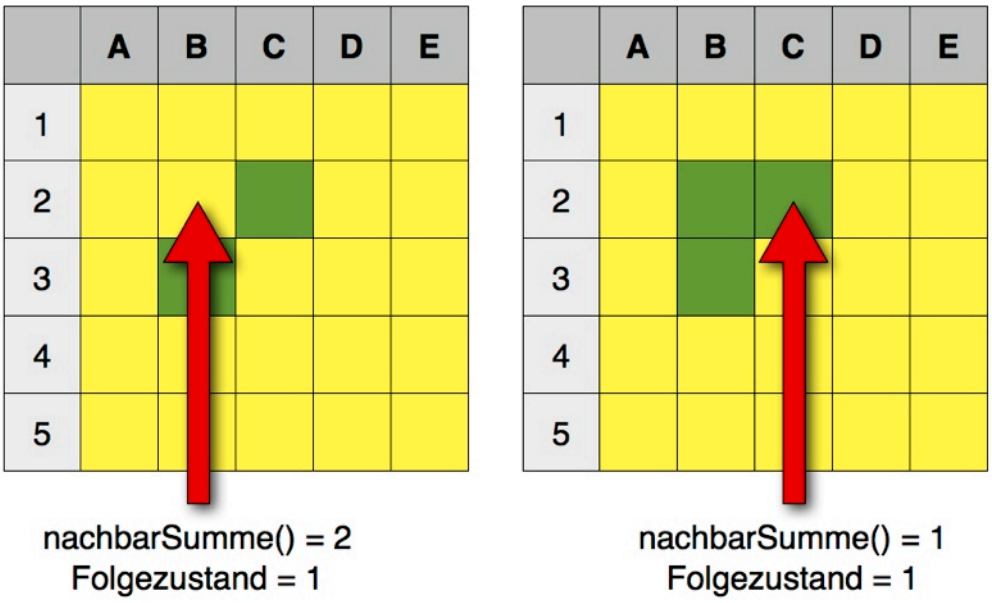
Wenn ein Feld weniger als zwei lebende Nachbarn hat, so ist es in der nächsten Generation tot. In dem Hilfs-Array **temp** wird das entsprechende Feld daher auf den Status `0` gesetzt. Hat das Feld genau zwei lebende Nachbarn, so wird der Status in der nächsten Generation in **temp** auf lebend (1) gesetzt. Wenn das Feld mehr als zwei lebende Nachbarn hat, so ist der Status im nächsten Zyklus wieder tot (0).

Wozu dient der Array temp, der hier benutzt wird?

Der Hilfsarray **temp** wird zusammen mit dem Hauptarray **feld** am Anfang der Klasse **Spielfeld** deklariert:

```
   int[][] feld, temp;
```

Um den Status eines Einzelfeldes in der *Folgegeneration* zu berechnen, werden die Zustände der vier Nachbarfelder in der *aktuellen* Generation ausgewertet. Würde nun der Zustand des gerade betrachteten Einzelfeldes sofort im Array **feld** geändert, so wäre die Berechnung für den rechten und den unteren Nachbarn dieses Einzelfeldes nicht mehr korrekt.



9.3-7 Eine Fehlberechnung

Dieses Bild zeigt, was passieren würde, wenn man den Status des Feldes B2 sofort nach der Berechnung verändern würde. Im rechten Teil des Bildes sieht man, dass jetzt der Status des Feldes C2 falsch berechnet wird. Da B2 bereits den neuen Folgezustand 1 angenommen hat, hat das Feld C2 plötzlich einen lebenden Nachbarn und nimmt daher den falschen Folgezustand 1 an. An sich hat das Feld C2 in der aktuellen Generation keinen lebenden Nachbarn (siehe linkes Teilbild), daher müsste der Folgezustand von C2 = 0 sein.

Um den Status der Nachbarn zu berechnen, darf das Einzelfeld noch nicht seinen Status ändern, sondern muss den alten Status behalten. Der neue Status des Einzelfeldes wird vorübergehend in dem Hilfsarray **temp** gespeichert. Erst wenn die Berechnung für alle Felder abgeschlossen ist, werden alle Felder auf den berechneten und in temp zwischengespeicherten Status gesetzt:

```
    for (int y = 0; y < 50; y++)
        for (int x = 0; x < 50; x++)
           feld[x][y] = temp[x][y];
```

Übungen

Übung 9.3 - 1 (2 Punkte)

Schreiben Sie jetzt die noch fehlende Methode **nachbarSumme()**, die die Zahl der lebenden Nachbarn eines Feldes zurück liefert. Jedes Feld hat genau vier Nachbarn: links, rechts, oben und unten, und die Rand- und Eckfelder sind mit den Feldern der gegenüberliegenden Seite verknüpft, wie bei einer Torus-Welt.

Übung 9.3 - 2 (3 Punkte)

Statten Sie das Applet mit einem Button "nächste Generation" oder "nächster Zyklus" aus. Wenn dieser Button geklickt wird, soll die Methode **Spielfeld.neuerZyklus()** ausgeführt werden. Durch die **paint()**-Methode des Applets wird die Welt dann wieder neu gezeichnet. Vergessen Sie nicht den Aufruf von **repaint()** in der **ActionPerformed()**-Methode!

Übung 9.3 - 3 (2 Punkte)

Ergänzen Sie das Applet um einen zweiten Button "neues Spiel". Wenn dieser Button geklickt wird, so startet ein neues Spiel mit einer neuen Zufalls-Belegung der Felder.

Exkurs: Eine bessere Berechnungsmöglichkeit

Bisher hatten wir den Zustand der Felder in der nächsten Generation wie folgt berechnet:

```
        for (int y = 0; y < 50; y++)
          for (int x = 0; x < 50; x++)
          {
              int s = nachbarSumme(x,y);
              if (s < 2)
                 temp[x][y] = 0;
              else if (s < 3)
                 temp[x][y] = 1;
              else if (s <= 4)
                 temp[x][y] = 0;
          }
```

Zunächst wurde mit **nachbarSumme()** die Zahl der lebenden Nachbarn ermittelt, dann kam eine verschachtelte if-else-Anweisung, mit deren Hilfe der nächste Zustand entschieden wurde.

Betrachten Sie nun eine alternative Möglichkeit:

```
        for (int y = 0; y < 50; y++)
          for (int x = 0; x < 50; x++)
              temp[x][y] = zustand[nachbarSumme(x,y)];
```

Diese drei Zeilen leisten genau das Gleiche wie der obige Quelltext. In einem neuen Array zustand, der aus genau fünf int-Zahlen besteht, werden bei der Deklaration / Initialisierung die Folgezustände gespeichert:

```
  int[]   zustand = {0,1,1,1,0};
```

Diese Anweisung erfolgt am Anfang der Klasse Spielfeld bei der Deklaration der Attribute. Dieser Array ist nichts anderes als die Umsetzung der Tabelle mit der Übergangsfunktion:

Zahl der lebenden Nachbarn	Neuer Zustand
0	0
1	1
2	1
3	0
4	0

***9.3-8** Die Übergangsfunktion*

Die Zahl der lebenden Nachbarn wird einfach als Index der Arrayelemente interpretiert, der neue Zustand ist dann der jeweilige Wert.

Schritt 4 - ein Timer

Es ist schon sehr lästig, wenn man jedes Mal auf den Button "nächster Zyklus" klicken muss, um die nächste Generation der Felder zu berechnen. Viel schöner wäre es, wenn das Applet automatisch alle 200 Millisekunden oder alle 100 Millisekunden eine neue Generation berechnen und zeichnen würde.

In Java geht das ohne Weiteres, man muss das Applet dazu mit einem **Timer** ausstatten. Die erforderlichen Komponenten finden wir in der Klasse **Runnable**, die wir in das Applet einbinden müssen:

```
public class Anzeige extends JApplet implements ActionListener, Runnable
```

Der Timer ist ein Attribut der Klasse **Thread**. Unter einem **Thread** versteht man einen Teil eines Prozesses, der gemeinsam mit anderen Threads vom Betriebssystem ausgeführt wird. Jeder Thread besitzt seinen eigenen Befehlszähler und seinen eigenen Stack, auf dem wichtige Informationen, zum Beispiel Zwischenergebnisse, abgelegt werden können. Weitere Erklärungen würden hier zu weit führen.

Die Klasse **Runnable** stellt zwei wichtige Methoden zur Verfügung, die in dem Applet auf jeden Fall *überschrieben* werden müssen. **Runnable** ist eine *abstrakte* Klasse, stellt also nur *leere Methoden* bereit, die noch mit Inhalt gefüllt werden müssen. Diese beiden Methoden heißen **start()** und **run()**. Die früher oft verwendete Methode **stop()** wird in aktuellen Java-Versionen nicht mehr verwendet.

Schauen wir uns zunächst die Deklaration des Timers an:

```
Thread uhr;
```

Das war ja kurz. Für die Initialisierung des Timers benötigen wir jetzt die **start()**-Methode des Applets, die wir bisher ja immer gelöscht hatten, wenn BlueJ für uns ein neues Applet anlegte:

```
public void start()
{
  if (uhr == null);
  {
     uhr = new Thread(this);
     uhr.start();
  }
}
```

Falls also in **start()** festgestellt wird, dass noch kein Timer initialisiert wurde `(uhr == null)`, so wird der Timer initialisiert und gestartet.

Betrachten wir nun die **run()**-Methode, die wir ebenfalls neu anlegen müssen:

```
public void run()
{
   while (true)
   {
      welt.neuerZyklus();
      repaint();
      try { Thread.sleep(200); }
      catch (InterruptedException e) { }
   }
}
```

Das zentrale Element der **run()**-Methode ist eine endlose while-Schleife; die Bedingung `while(true)` ist immer erfüllt, daher läuft die while-Schleife ständig.

Im Wesentlichen werden in der Schleife drei Anweisungen ausgeführt:

1. Zunächst wird **Spielfeld.neuerZyklus()** aufgerufen und damit die nächste Generation berechnet.
2. Mit dem **repaint()**-Befehl wird das neue Spielfeld vom Applet gezeichnet.
3. Und mit der **Thread.sleep()**-Anweisung wird der Thread für eine kurze Zeitspanne (hier 200 Millisekunden) angehalten.

Der Aufruf der **sleep()**-Anweisung kann allerdings zu Problemen führen, vor allem, wenn mehrere Threads vom Betriebssystem gleichzeitig ausgeführt werden, was ja fast immer der Fall ist. Daher *versucht* das Programm mit `try` diese Anweisung auszuführen. Wenn das gelingt, ist alles gut. Wenn nicht, wird im `catch`-Zweig eine interne Fehlerroutine aufgerufen.

Nach 200 ms wird ein neuer Schleifendurchgang gestartet, eine neue Generation berechnet und gezeichnet. Dann wird wieder 200 ms gewartet, bis die Schleife erneut gestartet wird. Erst wenn der Benutzer das Java-Applet beendet, wird die while-Schleife gestoppt.

Übungen

Übung 9.3 - 7 (4 Punkte)

Bauen Sie die besprochenen Methoden in Ihr Applet ein, geben Sie den Buttons sinnvolle Namen und beobachten Sie den Verlauf des Automaten über mehrere Hundert Zyklen.

Übung 9.3 - 8 (2 Punkte)

Zeigen Sie mit dem **g.drawString()**-Befehl ständig die Zahl der bereits zurückgelegten Zyklen im Applet an! Achten Sie auf eine lesbare Ausgabe der Zahl!

Übung 9.3 - 9 (2 Punkte)

Probieren Sie andere (komplexere) Übergangsfunktionen in der Klasse **Spielfeld** aus und dokumentieren Sie Ihre Ergebnisse zum Beispiel durch Screenshots oder besser durch eine Präsentation vor dem Kurs.

Übung 9.3 - 10 (2 Punkte)

Experimentieren Sie mit mehr als zwei Zuständen, zum Beispiel "lebendig", "krank", "tot".

Übung 9.3 - 11 (2 Punkte)

Experimentieren Sie mit mehr als vier Nachbarn, indem Sie die diagonalen Nachbarn ebenfalls mit berücksichtigen.

Übung 9.3 - 12 (2 Punkte)

Berücksichtigen Sie in Ihrer komplexeren Übergangsfunktion den bisherigen Zustand der jeweiligen Zelle. Lebende Zellen sollen sich anders verhalten als tote Zellen.

Damit ist der Kurs "Einführung in das Programmieren mit Java" für die Stufe EF beendet. Die Folge 9 werden viele EF-Kurse wahrscheinlich nicht mehr schaffen, was aber auch nicht schlimm ist; man kann die Folge 9 auch sehr gut als Einstieg für die Stufe Q1 verwenden.

Anhang

Kompetenzbereiche und Inhaltsfelder des Faches Informatik

nach dem Kernlehrplan für die Sekundarstufe II Gymnasium / Gesamtschule in NRW von 2013, gültig ab dem 1. August 2014, beginnend mit der Einführungsphase.

Kompetenzbereiche

A - Argumentieren

Informatische Zusammenhänge, Vorgehensweisen, Lösungsansätze und Entwurfsentscheidungen bedürfen der Erläuterung und Begründung, um Transparenz, Nachvollziehbarkeit und Überprüfbarkeit im Diskurs zu gewährleisten. Argumentieren umfasst das Erläutern, Begründen und Beurteilen in informatischen Sachzusammenhängen und Prozessen. Erläutern bedeutet, einen Sachverhalt zu veranschaulichen und verständlich zu machen. Unter Begründen wird die Darlegung von rational nachvollziehbaren Argumenten auf der Grundlage von Begriffen, Regeln, Methoden und Verfahren der Informatik verstanden. Dazu gehört auch, den Begründungszusammenhang durch geeignete Beispiele zu veranschaulichen. Beurteilen meint, zu einem informatischen Sachverhalt oder Prozess ein selbstständiges Urteil unter Verwendung von Fachwissen und Fachmethoden zu formulieren und zu begründen. Argumentieren umfasst auch die Bewertung von Nutzen, Grenzen und Auswirkungen von Informatiksystemen.

Kompetenzen am Ende der EF

Die Schülerinnen und Schüler

- erläutern und begründen methodische Vorgehensweisen, Entwurfs- und Implementationsentscheidungen sowie Aussagen über Informatiksysteme,
- analysieren und erläutern informatische Modelle,
- analysieren und erläutern Computerprogramme,
- beurteilen die Angemessenheit informatischer Modelle.

M - Modellieren

Um ein Problem aus einem inner- oder außerinformatischen Kontext lösen zu können, wird in der Regel zunächst ein informatisches Modell entwickelt, das auf einem prozessorgesteuerten Gerät implementiert werden kann. Informatisches Modellieren zielt auf eine abstrahierende Beschreibung der wesentlichen Komponenten und Parameter eines realen oder geplanten Systems sowie des Ordnungsgefüges und der Wirkungsbeziehungen zwischen ihnen. Der Modellierungsprozess beginnt mit der Analyse und einer strukturierten Zerlegung des Ausgangsproblems. Teilkomponenten müssen identifiziert, konstruiert und gegebenenfalls miteinander vernetzt werden. Ein Ergebnis eines Modellierungsprozesses ist in der Regel eine formale, textuelle oder grafische Darstellung.

Kompetenzen am Ende der EF

Die Schülerinnen und Schüler

- konstruieren zu kontextbezogenen Problemstellungen informatische Modelle,
- modifizieren und erweitern informatische Modelle.

I - Implementieren

Implementieren umfasst die Umsetzung eines Modells in ein Informatiksystem. Dazu gehören das Programmieren, Evaluieren und Validieren von Modellbestandteilen unter Nutzung geeigneter Werkzeuge. Grundlegende Methoden und Denkweisen der Programmentwicklung werden dabei in den Vordergrund gestellt. Die Programmerstellung ist ein bedeutsamer Bestandteil des Problemlösungsprozesses, weil erst dadurch das Modell wirksam wird. An dem entstandenen Informatiksystem können Wirkungen der Modellentscheidungen diskutiert sowie Ursachen und Tragweite von möglichen Fehlern im Modell erkannt und korrigiert werden. Dadurch werden die Selbstreflexion des Lösungsprozesses und eine vertiefte Modellkritik unterstützt.

Kompetenzen am Ende der EF

Die Schülerinnen und Schüler

- implementieren auf der Grundlage von Modellen oder Modellausschnitten Computerprogramme,
- modifizieren und erweitern Computerprogramme,
- testen und korrigieren Computerprogramme.

D - Darstellen und Interpretieren

Die Informatik hat zur Unterstützung von Problemlöse- und Modellbildungsprozessen ein reiches Repertoire an Darstellungsformen entwickelt. Schülerinnen und Schüler werden nach und nach mit unterschiedlichen Darstellungsformen konfrontiert, die sie in inner- und außerinformatischen Kontexten selbst nutzen. Vorgegebene Darstellungen müssen anwendungsbezogen interpretiert werden. Im Rahmen eigener Problemlösungen müssen angemessene Darstellungsformen unter Verwendung der fachspezifischen Notation angewendet werden. Dies fördert ein Verständnis von Zusammenhängen und Bezügen zwischen unterschiedlichen informatischen Sachverhalten sowie die Fähigkeit, diese anderen deutlich zu machen.

Kompetenzen am Ende der EF

Die Schülerinnen und Schüler

- interpretieren Daten und erläutern Beziehungen und Abläufe, die in Form von textuellen und grafischen Darstellungen gegeben sind,
- überführen gegebene textuelle und grafische Darstellungen informatischer Zusammenhänge in die jeweils andere Darstellungsform,
- stellen informatische Modelle und Abläufe in Texten, Tabellen, Diagrammen und Grafiken dar.

K - Kommunizieren und Kooperieren

Die Kenntnis und Nutzung arbeitsteiliger und kooperativer Vorgehensweisen ist für die Entwicklung komplexer Informatiksysteme erforderlich, um prozessorientiertes Arbeiten zu planen und abzusichern. Zum Kommunizieren im Sinne eines fachlichen Austausches gehören die sachadäquate Darstellung und Dokumentation zur Weitergabe von Sachverhalten sowie die Nutzung geeigneter Werkzeuge, die die Kommunikation unterstützen. Für eine sachangemessene und präzise Verständigung über informatische Gegenstände sind ein angemessener Umgang mit Fachbegriffen und der sukzessive Aufbau einer Fachsprache unerlässlich.

Kompetenzen am Ende der EF

Schülerinnen und Schüler

- verwenden Fachausdrücke bei der Kommunikation über informatische Sachverhalte,
- kommunizieren und kooperieren in Gruppen und in Partnerarbeit,
- präsentieren Arbeitsabläufe und -ergebnisse.

Inhaltsfelder

I1 - Daten und ihre Strukturierung

Die automatische Verarbeitung von Informationen mittels Maschinen ist überhaupt erst durch deren digitale Repräsentation in Form von Daten möglich. Für die rechnergestützte Lösung von Problemen in inner- und außerinformatischen Kontexten müssen daher Informationen in angemessener Struktur durch Daten und zugehörige Operationen repräsentiert werden, so dass die Daten zielgerichtet und effizient automatisch verarbeitet und die Ergebnisse wiederum als Information interpretiert werden können.

Inhaltlicher Schwerpunkt: Objekte und Klassen

I2 - Algorithmen

Zu vielen bedeutenden wissenschaftlichen Erfolgen und technischen Errungenschaften der jüngeren Zeit hat die Informatik maßgeblich beigetragen. Neben der rasanten Steigerung der Leistungsfähigkeit der technischen Systeme sind diese Fortschritte insbesondere der Entwicklung von innovativen und effizienten Algorithmen zu verdanken. Ein Algorithmus ist eine genaue Beschreibung von Handlungsschritten zur Lösung eines Problems, die von einem Prozessor ausgeführt werden können. Häufig verwendete Grundkonstrukte von Algorithmen sowie Algorithmen, die im Kontext bestimmter Problemklassen von elementarer Bedeutung sind, lassen sich unter Berücksichtigung ihrer Effizienz adaptieren, um neue Aufgabenstellungen in konkreten Anwendungskontexten problemgerecht einer automatischen Verarbeitung zuzuführen.

Inhaltlicher Schwerpunkt: Analyse, Entwurf und Implementierung einfacher Algorithmus, Algorithmus zum Suchen und Sortieren.

I3 - Formale Sprachen und Automaten

Der Einsatz von Informatiksystemen zur Lösung komplexer Probleme ist nur unter Verwendung formaler Sprachen als Mittler zwischen Mensch und Maschine möglich. Sprachen dienen zur Kommunikation und genügen Regeln zur Bildung von Wörtern und Sätzen. Formale Sprachen der Informatik werden durch Grammatiken präzise beschrieben. Zu formalen Sprachen können Automaten entwickelt werden, die die Wörter der Sprache akzeptieren oder weiterverarbeiten. Eine fachliche Beschreibung von Automaten mithilfe einer Menge von Zuständen samt Regeln für die zeitliche Abfolge von Zustandsübergängen ist als Modellierungstechnik in verschiedenen Problemfeldern anwendbar. Automaten eigenen sich in besonderem Maße, um mithilfe theoretischer Betrachtungen auch die Grenzen von Automatenmodellen zu beleuchten.

Inhaltlicher Schwerpunkt: Syntax und Semantik einer Programmiersprache

Das Thema "Formale Sprachen und Automaten" verwirrt jeden gestandenen Informatiklehrer zunächst. Bisher wurden formale Sprachen und Automaten am Ende des Informatikunterrichts in der Stufe 13 bzw. Q2 behandelt, zum Beispiel beim Thema "Compilerbau". Soll das Thema jetzt in die EF vorverlagert werden? Schauen wir uns einmal die Ausführungen im Kernlehrplan an, was in der EF zu diesem Thema eigentlich gemacht werden soll:

Die Schülerinnen und Schüler

- implementieren einfache Algorithmen unter Beachtung der Syntax und Semantik einer Programmiersprache,
- interpretieren Fehlermeldungen und korrigieren den Quellcode.

Das Ganze erscheint mir hier wie ein guter Witz! Toll, wie hier alter Wein in neue, wohlklingende Schläuche verpackt wird. Was machen die S. denn anderes als Algorithmen zu implementieren und nach Fehlern zu suchen, wenn sie einfache Problemstellungen am PC programmieren? Offensichtlich glauben die Macher des Kernlehrplans, dass es Schulen gibt, wo Programme geschrieben werden ohne Beachtung der Syntax und Semantik einer Programmiersprache und ohne nach Fehlern zu suchen. Echt toll. Noch schlimmer allerdings wäre es, wenn die Macher des Kernlehrplans damit auch noch Recht hätten, dass es also tatsächlich solchen Informatikunterricht in Deutschland gibt, wo nichts implementiert wird.

I4 - Informatiksysteme

Informatiksysteme sind heute weltweit miteinander vernetzt. Ein Informatiksystem ist eine spezifische Zusammenstellung von Hardware-, Software- und Netzwerkkomponenten zur Lösung eines Anwenderproblems. Gegenstand der Betrachtung in diesem Inhaltsfeld sind schwerpunktmäßig der prinzipielle Aufbau singulärer und vernetzter Rechnersysteme und deren Interaktion untereinander und mit dem Benutzer.

Inhaltlicher Schwerpunkt: Digitalisierung, Einzelrechner, Dateisystem, Internet

I5 - Informatik, Mensch und Gesellschaft

Informatiksysteme stehen in intensiver Wechselwirkung mit Individuum und Gesellschaft. Ihr Einsatz hat weitreichende Konsequenzen für unsere Lebens- und Arbeitswelt. Handlungsspielräume müssen im Spannungsfeld von Rechten und Interessen des Individuums, gesellschaftlicher Verantwortung und möglichen Sicherheitsrisiken wahrgenommen werden.

Inhaltlicher Schwerpunkt: Einsatz von Informatiksystemen, Wirkung der Automatisierung, Geschichte der automatischen Datenverarbeitung.

Hier muss ich wieder zugeben, dass man in meinem Skript so gut wie nichts davon lesen kann. Das ist aber auch nicht meine Aufgabe, denn mein Skript will kein gutes Schulbuch der Informatik ersetzen, sondern höchstens ergänzen. Es obliegt hier dem einzelnen Lehrer, seinen Informatikunterricht entsprechend zu ergänzen.