Programmieren mit Java

Eine Einführung ins Programmieren mit Java, NetBeans und Jeda

1 Erste Programmierschritte

1.1 Einleitung

Um ein eigenes Programme zu erstellen, müssen die Anweisungen an den Computer in einer ***Programmiersprache*** formuliert werden. Oft werden dabei vorgefertigte Befehlssammlungen, sogenannte ***Bibliotheken***, verwendet. Das geschriebene Programm muss anschliessend geprüft, meist korrigiert und getestet werden. Für alle diese Aufgaben wird fast immer eine ***Entwicklungsumgebung*** eingesetzt.

1.1.1 Programmiersprache

Eine Programmiersprache ist eine spezielle Sprache, die zur Erstellung von Computerprogrammen verwendet wird. Programmierer formulieren das Programm als ***Quelltext***. Dieser wird automatisiert in ***Maschinencode*** übersetzt, welcher von Computern ausgeführt werden kann.

1.1.2 Entwicklungsumgebung

Eine Entwicklungsumgebung ist ein Programm, welches sämtliche wichtigen Funktionen für die Softwareentwicklung zur Verfügung stellt. Dazu gehören:

- Projektverwaltung
- Bearbeiten, Überprüfen und Übersetzen der Quelltexte
- Fehlersuche
- Zusammenarbeit mit anderen Entwicklern

Die in diesem Kurs verwendete Entwicklungsumgebung heisst ***NetBeans***. Sie unterstützt verschiedene Programmiersprachen, darunter auch Java. NetBeans ist Open-Source-Software und wird von Oracle/Sun entwickelt.

1.1.3 Bibliotheken

Bibliotheken sind vorgefertigte Programmteile, welche dem Programmierer viele Aufgaben erleichtern, zum Beispiel die Ausgabe von Text und Grafik, das Abspielen von Audiodaten oder das Abfragen von Benutzereingaben.

Mit der Programmiersprache Java wird bereits eine umfangreiche Bibliothek mitgeliefert. Für diesen Kurs wird eine zusätzliche Bibliothek namens ***Jeda*** verwendet, welche aus didaktischen Gründen möglichst einfach gestaltet worden ist.

Aufgabe 1.1

Um Jeda in NetBeans verwenden zu können, wird das Jeda NetBeans-Plugin benötigt. Laden Sie das Plugin vom Kiwiki herunter und installieren Sie es.

1.2 Erstes Java-Projekt

1.2.1 Projekt erstellen

Im Menü ***File*** klicken Sie auf ***New Project…*** und wählen die Kategorie ***Jeda*** aus. Klicken Sie auf ***Next*** und wählen Sie einen sinnvollen Namen (z.B. `BeispielProjekt`) und Speicherort für das Projekt. Nun können Sie den Vorgang mit ***Finish*** abschliessen.

Anschliessend müssen Sie eine Quellcode-Datei in das Projekt Einfügen. Klicken Sie dazu im Menü ***File*** klicken Sie auf ***New File…*** und wählen die Kategorie ***Jeda*** und den Dateityp ***Jeda Program Class*** aus. Klicken Sie auf ***Next*** und wählen Sie einen sinnvollen Namen für die Datei (z.B. `BeispielProgramm`). Nun können Sie den Vorgang mit ***Finish*** abschliessen.

1.2.2 Aufbau des Projekts

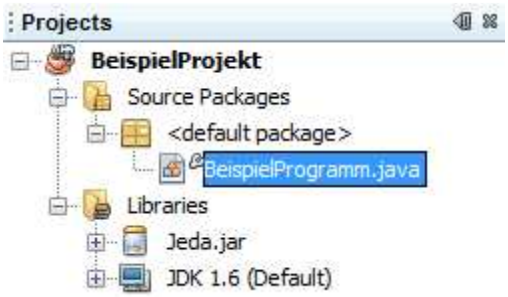
Wenn das Projekt ***MyFirstProgram*** erstellt worden ist, wird es in der Projektübersicht von NetBeans angezeigt. Unterhalb des Projekts werden folgende zwei Ordner angezeigt:

- ***Source Packages*** enthält den Quellcode des Programms.
- ***Libraries*** enthält Verweise auf die benötigten Bibliotheken.

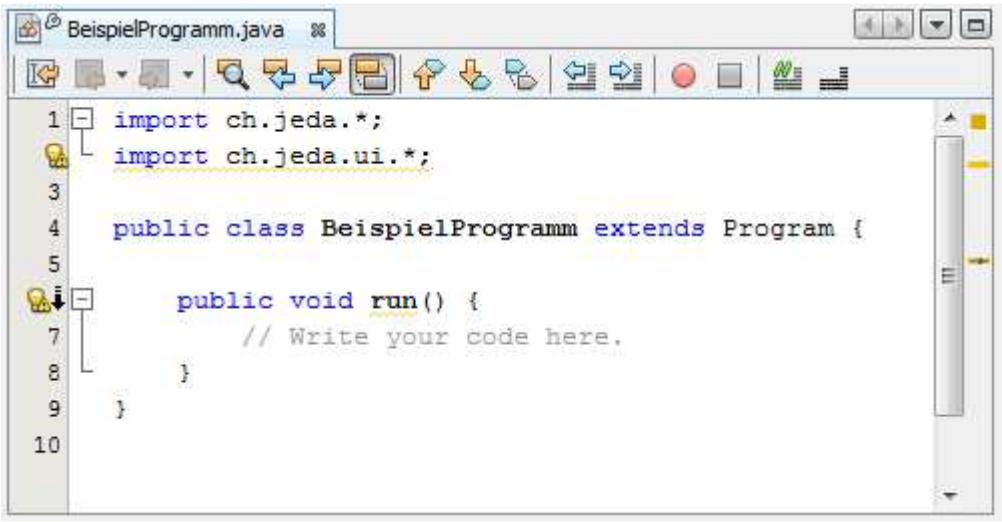
Alle Java-Projekte verwenden die Java-Bibliothek JDK. Unsere Jeda-Projekte zusätzlich die Jeda-Bibliothek.

Der Quellcode eines Java-Programms ist in ***Pakete*** (*packages*) gegliedert, diese in ***Quellcode-Dateien***. Sie haben die Dateiendung `.java` und enthalten ***Klassen***, die Grundbausteine eines Java-Programms.

Vorläufig arbeiten wir ohne Pakete, wir verwenden nur das vorgegebene ***default package***. Es sollte schon die vorhin erzeugte Quellcode-Datei `BeispielProgramm` enthalten:



Durch einen Doppelklick auf die Datei `BeispielProgramm.java` wird sie im Editor geöffnet:



1.2.3 Aufbau der Java-Klasse

Der schon vorhandene Quelltext stellt das Gerüst für ein einfaches Programm dar. Die genaue Bedeutung der einzelnen Befehle werden wir später besprechen, hier nur eine kurze Erklärung:

```
import ch.jeda.*;
import ch.jeda.ui.*;
```

Diese zwei Zeilen bedeuten, dass in dieser Klasse die Pakete `ch.jeda` und `ch.jeda.ui` aus der Jeda-Bibliothek verwendet werden können.

```
public class BeispielProgram extends Program {
```

Diese Zeile besagt, dass diese Klasse `BeispielProgram` heisst und ein ausführbares Jeda-Programm darstellt. Auch hier muss der Name der Klasse mit dem Namen in der Projektansicht übereinstimmen.

```
    public void run() {
```

Diese Zeile besagt, dass anschliessend die ***Anweisungen*** folgen, welche ausgeführt werden sollen, wenn das Programm gestartet wird. Eine Klasse kann mehrere solche Gruppen von Anweisungen enthalten, sie werden ***Methoden*** genannt.

```
        // Write your code here.
```

Was hinter einem doppelten Schrägstrich steht, gilt als ***Kommentar*** und wird vom Computer nicht als Teil des Programms betrachtet. Dieser Kommentar soll den Programmierer darauf hinweisen, wo er die Anweisungen hinschreiben soll.

```
    }
}
```

Die Methode und die Klasse werden mit einer schliessenden geschweiften Klammer abgeschlossen.

Aufgabe 1.2

Nun möchten Sie sicher Ihr erstes Programm schreiben. Es soll ein leeres Fenster öffnen. Ersetzen Sie dazu den Kommentar durch die Anweisung:

```
new Window();
```

Führen Sie das Programm durch drücken von F6 aus.

Gestalten Sie anschliessend den Quellcode individueller, indem Sie:

- die Klasse umbenennen (im Projektfenster Ctrl+R drücken).
- einen eigenen Kommentar einfügen.

1.3 Programmieren

1.3.1 Java-Grundregeln

Wie eine gewöhnliche Sprache hat auch eine Programmiersprache eine Grammatik. Der Programmierer muss sich streng daran halten, ansonsten versteht der Computer nicht, was gemeint ist. Die wichtigsten Regeln von Java sind:

- Zu jeder öffnenden Klammer gehört eine entsprechende schliessende Klammer. In Java wird neben der runden Klammer `( )` auch die geschweifte Klammer `{ }` verwendet.
- Jeder Befehl wird mit einem Semikolon `;` abgeschlossen.

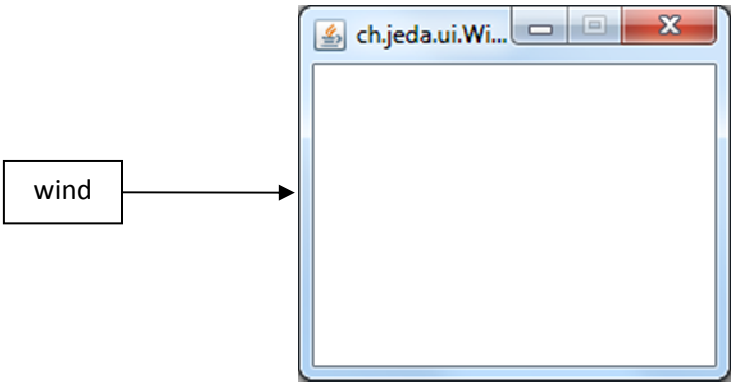
- Namen dürfen nur aus Buchstaben, Ziffern und dem Unterstrich _ bestehen. Am Anfang darf keine Ziffer stehen.
- Was nach einem Doppelschrägstrich `//` steht, ist ein Kommentar und wird ignoriert.
- Gross- und Kleinschreibung ist wichtig: `Beispiel` ist nicht dasselbe wie `beispiel`.

1.3.2 Variablen

Wir haben schon die Anweisung

```
new Window();
```

kennengelernt und festgestellt, dass damit ein Fenster erzeugt wird. Nun möchten wir auf das Fenster zeichnen. Damit aber der Computer weiss, auf welches Fenster wir zeichnen möchten, benötigen wir einen Verweis auf das von uns erzeugte Fenster. In Java dienen ***Variablen*** als solche Verweise. Wir möchten also eine Variable, welche auf unser Fenster verweist:



Die entsprechende Java-Anweisung sieht so aus:

```
Window wind = new Window();
```

Hier offenbart sich schon eine Besonderheit von Programmiersprachen. Das Gleichheitszeichen bedeutet eine ***Zuweisung*** von rechts nach links. Folgendes passiert:

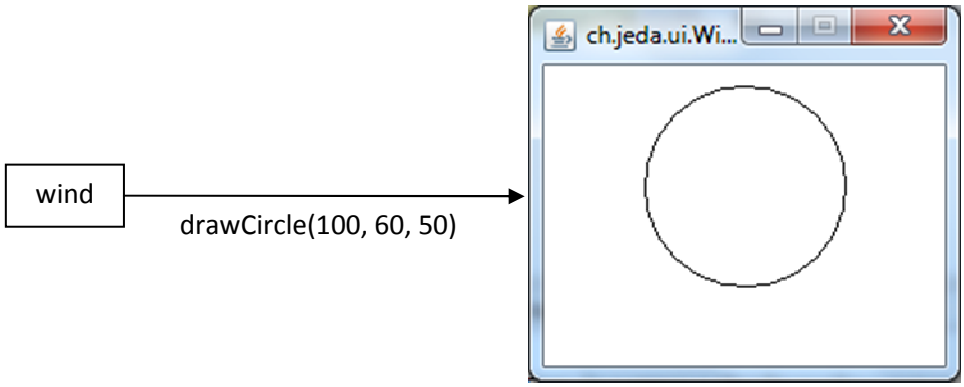
- Der Teil links vom Gleichheitszeichen bedeutet, dass wir eine Variable `wind` definieren, welche auf (irgendein) Fenster verweisen kann.
- Rechts vom Gleichheitszeichen wird bekannterweise ein neues Fenster erzeugt.
- Das Gleichheitszeichen bewirkt, dass das neu erstellte Fenster der Variablen `wind` zugewiesen wird.

1.3.3 Methoden

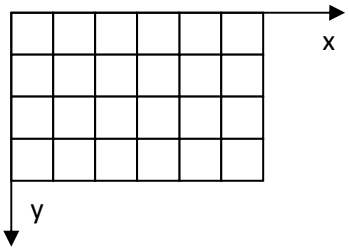
Nun können wir die Variable `wind` benützen, um dem Fenster Zeichenbefehle zu übermitteln. Zum Beispiel kann so ein Kreis in das Fenster gezeichnet werden:

```
wind.drawCircle(100, 60, 50);
```

Diese Anweisung bedeutet, dass wir dem Fenster, auf welches die Variable `wind` verweist, den Befehl `drawCircle(100, 60, 50)` senden. Dies veranlasst das Fenster, einen Kreis mit Mittelpunkt (100, 60) und Radius 50 zu zeichnen.



1.3.4 Zeichnen



In der Computergrafik wird üblicherweise ein Koordinatensystem eingesetzt, welches auf der Anordnung der Grafikpunkte (***Pixel***) auf dem Bildschirm basiert. Dabei werden nur ganz Zahlen verwendet. Der Ursprung (0, 0) liegt in der oberen linken Ecke, die x-Achse zeigt nach rechts und die y-Achse nach unten.

Folgend werden einige Methoden erklärt:

`drawCircle(x, y, r)`
: Zeichnet einen Kreis mit Mittelpunkt (x, y) und Radius r.

`drawLine(x1, y1, x2, y2)`
: Zeichnet eine gerade Linie von (x1, y1) nach (x2, y2).

`drawRectangle(x, y, w, h)`
: Zeichnet ein Rechteck mit Breite w und Höhe h. Die linke obere Ecke liegt in (x, y).

`drawTriangle(x1, y1, x2, y2, x3, y3)`
: Zeichnet ein Dreieck mit den Eckpunkten (x1, y1), (x2, y2) und (x3, y3).

`fillCircle(x, y, r)`
: Zeichnet einen ausgefüllten Kreis mit Mittelpunkt (x, y) und Radius r.

`fillRectangle(x, y, w, h)`
: Zeichnet ein ausgefülltes Rechteck mit Breite w und Höhe h. Die linke obere Ecke liegt in (x, y).

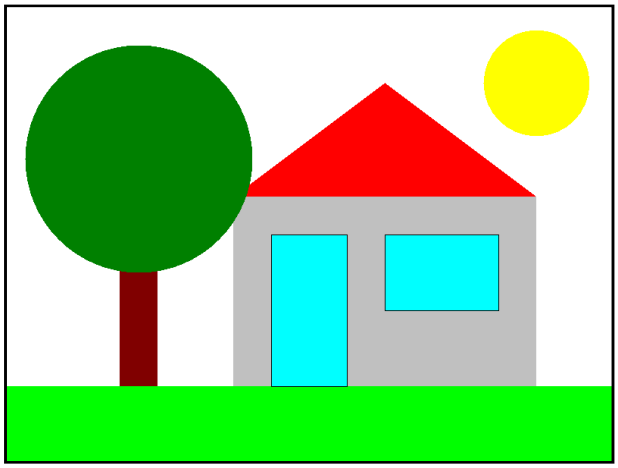
`fillTriangle(x1, y1, x2, y2, x3, y3)`
: Zeichnet ein ausgefülltes Dreieck mit den Eckpunkten (x1, y1), (x2, y2) und (x3, y3).

`setColor(c)`
: Setzt die Zeichenfarbe. Anstelle von c kann einer der folgenden Farbwerte angegeben werden:

Color.BLACK	Color.NAVY	Color.SILVER	Color.BLUE
Color.MAROON	Color.PURPLE	Color.RED	Color.FUCHSIA
Color.GREEN	Color.TEAL	Color.LIME	Color.AQUA
Color.OLIVE	Color.GRAY	Color.YELLOW	Color.WHITE

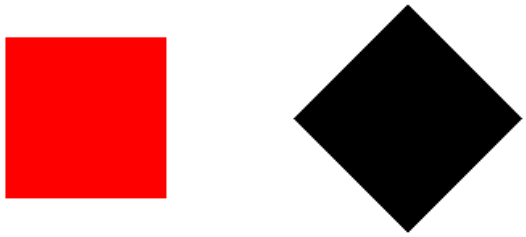
Aufgabe 1.3

Schreiben Sie ein Programm, welches das folgende Bild zeichnet:



Zusatzaufgabe 1.4

Schreiben Sie ein Programm, welches zwei Quadrate zeichnet, die so aussehen:



Zusatzaufgabe 1.5

Schreiben Sie ein Programm, welches das abgebildete gleichseitige Dreieck zeichnet. Die obere Ecke hat die Koordinaten x = 400 und y = 100, die Seitenlänge beträgt 200 Pixel.

2 Variablen und Datentypen

2.1 Darstellung von Werten

Alle in Variablen gespeicherten Werte werden im Speicher als Folge von Nullen und Einsen abgelegt.

2.1.1 Darstellung von ganzen Zahlen

Wollen wir im Speicher z.B. die Zahl 89 speichern, müssen wir sie ins Binärsystem umrechnen:

Zahl	Interpretation	Binär
89	$0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$	01011001
13	$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$	00001101

Wir nehmen an, dass wir eine 8 bit Speicherstelle zur Verfügung haben. In diesem Fall kann diese Speicherstelle eine Zahl zwischen 0 und 255 aufnehmen.

Sollen in dieser Speicherstelle auch negative Zahlen dargestellt werden können, könnten wir beispielsweise definieren, dass das höchstwertige Bit (ganz links) als Vorzeichen-Bit interpretiert wird: 0 heisst z.B. positiv, 1 heisst negativ. Unsere Speicherstelle kann nun die Zahlen zwischen -127 und +127.

Zahl	Interpretation	Binär
-63	1 für minus, $0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$	10111111
42	0 für plus, $0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$	00101010

2.1.2 Darstellung von Dezimalzahlen

Wollen wir in dieser Bit-Folge allerdings eine Dezimalzahl speichern, könnten wir uns überlegen, dass man jede Dezimalzahl wie folgt schreiben kann:

$$0.x \cdot 10^y$$

Das heisst, dass uns für die Mantisse x und den Exponenten y nur je 4 bit zur Verfügung stehen. Von diesen 4 bit brauchen wir wiederum jeweils das vorderste Bit als Vorzeichen-Bit.

Zahl	Interpretation	Binär
0.05	$0.5 \cdot 10^{-1}$, also 0 für plus, 101 für 5; 1 für minus, 001 für 1	01011001
-0.0006	$-0.6 \cdot 10^{-3}$, also 1 für minus, 110 für 6; 1 für minus, 011 für 3	11101011
50.0	$0.5 \cdot 10^2$, also 0 für plus, 101 für 5; 0 für plus, 010 für 2	01010010

2.1.3 Darstellung von Textzeichen

Aber auch Buchstaben und Sonderzeichen lassen sich mit 8 bit darstellen. Es gibt den ASCII-Standard (***A***merican ***S***tandard ***C***ode for ***I***nformation ***I***nterchange). Dort erhält jedes Zeichen einen Code zwischen 0 und 127 (oder zwischen 0 und 255 für Extended ASCII), was sich wiederum in unserem 8 bit Speicher ablegen lässt:

Zeichen	Code	Binär	Zeichen	Code	Binär	Zeichen	Code	Binär
Space	32	00100000	@	64	00100000	`	96	01100000
!	33	00100001	A	65	00100001	a	97	01100001
"	34	00100010	B	66	01000010	b	98	01100010
#	35	00100011	C	67	01000011	c	99	01100011
$	36	00100100	D	68	01000100	d	100	01100100
%	37	00100101	E	69	01000101	e	101	01100101
&	38	00100110	F	70	01000110	f	102	01100110
'	39	00100111	G	71	01000111	g	103	01100111
(	40	00101000	H	72	01001000	h	104	01101000
)	41	00101001	I	73	01001001	i	105	01101001
*	42	00101010	J	74	01001010	j	106	01101010
+	43	00101011	K	75	01001011	k	107	01101011
,	44	00101100	L	76	01001100	l	108	01101100
-	45	00101101	M	77	01001101	m	109	01101101
.	46	00101110	N	78	01001110	n	110	01101110
/	47	00101111	O	79	01001111	o	111	01101111
0	48	00110000	P	80	01010000	p	112	01110000
1	49	00110001	Q	81	01010001	q	113	01110001
2	50	00110010	R	82	01010010	r	114	01110010
3	51	00110011	S	83	01010011	s	115	01110011
4	52	00110100	T	84	01010100	t	116	01110100
5	53	00110101	U	85	01010101	u	117	01110101
6	54	00110110	V	86	01010110	v	118	01110110
7	55	00110111	W	87	01010111	w	119	01110111
8	56	00111000	X	88	01011000	x	120	01111000
9	57	00111001	Y	89	01011001	y	121	01111001
:	58	00111010	Z	90	01011010	z	122	01111010
;	59	00111011	[	91	01011011	{	123	01111011
<	60	00111100	\	92	01011100	|	124	01111100
=	61	00111101	]	93	01011101	}	125	01111101
>	62	00111110	^	94	01011110	~	126	01111110
?	63	00111111	_	95	01011111	Delete	127	01111111

Wollen wir also ein ganzes Wort speichern, benötigen wir gleich mehrere 8 bit Speicherstellen, nämlich eine pro Buchstabe.

Aufgabe 2.1

In unserer 8 bit Speicherstelle befindet sich die Bit-Folge 01110011. Was schliessen Sie daraus? Welche Zahl, welcher Buchstabe wurde gespeichert?

2.1.4 Datentypen

Ein Datentyp beschreibt, welche Art von Daten in einer Variablen gespeichert wird und wie viele Bytes Speicherplatz benötigt werden.

Wenn ein Programm auf einen gespeicherten Wert zugreift, muss es wissen, wie es die vorgefundenen Bits interpretieren soll, also welcher Datentyp der Wert hat. Dazu wird der ***Datentyp*** oder einfach ***Typ*** einer Variablen im Quelltext des Programms definiert.

```
Window
```

2.1.5 Variablendeklaration

Bereits im ersten Kapitel haben wir eine Variable verwendet und Java mitgeteilt, was in dieser Variablen gespeichert wird:

```
Window wind = new Window();
```

Diese Zeile kann man auch wie folgt schreiben (beide Schreibweisen sind gleichwertig):

```
Window wind;
wind = new Window();
```

Auf der ersten Zeile steht die sogenannte ***Variablendeklaration***. Wir deklarieren, dass die Variable `wind` auf ein `Window` verweisen soll. In der zweiten Zeile teilen wir Java dann mit, dass `wind` auf das soeben neu erstellte Fenster zeigen soll (Definition oder Zuweisung).

Die oben erwähnten Beispiele zeigen, dass der Computer die Nullen und Einsen im Speicher verschieden interpretieren kann. Aus diesem Grund ist es nötig, für jede Variable anzugeben, welcher Datentyp darin gespeichert wird.

Bevor eine Variable verwendet werden kann, muss sie deklariert werden. Anschliessend können wir der Variablen beliebig oft einen Wert des deklarierten Typs zuweisen.

Weitere Beispiele:

```
int anzahl;
anzahl = 13;
```

2.2 Ganze Zahlen in Java

2.2.1 Datentypen

Da Computer aber nicht unendlich viel Speicherplatz haben, können nur Zahlen in einem bestimmten Bereich verwendet werden. Java definiert folgende vier Datentypen für ganze Zahlen:

Datentyp	Grösse	kleinste Zahl	grösste Zahl
`byte`	8 bit	$-2^7 = -128$	$2^7 - 1 = 127$
`short`	16 bit	$-2^{15} = -32768$	$2^{15} = 32767$
`int`	32 bit	$-2^{31} = -2147483648$	$2^{31} = 2147483647$
`long`	64 bit	$-2^{63} = -9223372036854775808$	$2^{63} = 9223372036854775807$

Für unseren Kurs ist `int` der wichtigste Datentyp zur Speicherung von ganzen Zahlen. Der abgedeckte Bereich reicht für die meisten Anwendungen völlig aus.

2.2.2 Rechenoperationen

Natürlich können in Java auch die grundlegenden mathematischen Operationen mit ganzen Zahlen durchgeführt werden. Die folgende Tabelle bietet einen Überblick:

Operation	Mathematisch	Java
Addition	$a + b$	`a + b`
Subtraktion	$a - b$	`a - b`
Multiplikation	$a \cdot b$	`a * b`
Ganzzahl-Division	$a \div b$	`a / b`
Modulo (Rest der Ganzzahl-Division)	$a \bmod b$	`a % b`
Betrag	$\lvert a \rvert$	`Math.abs(a)`

Als Operanden können sowohl bekannte Variablennamen als auch Zahlen verwendet werden.

```
int a = 13;
int b = a / 2;       // ist gleich 6, nicht 6.5 !!!
int rest = a % 2;    // Rest ist gleich 1
```

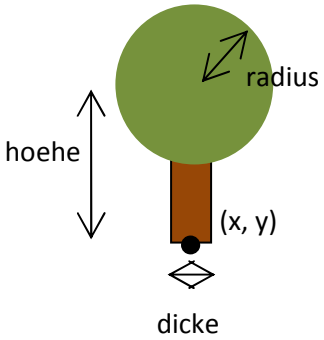
Es können auch mehrere Operationen nach den üblichen Distributivgesetzen verknüpft werden. Wo nötig können Klammern verwendet werden:

```
int t = Math.abs(-5) + 20 * x;
int u = x * (x - 1);
int v = Jeda.randomInt(100);    // liefert eine Zufallszahl < 100
```

Aufgabe 2.2

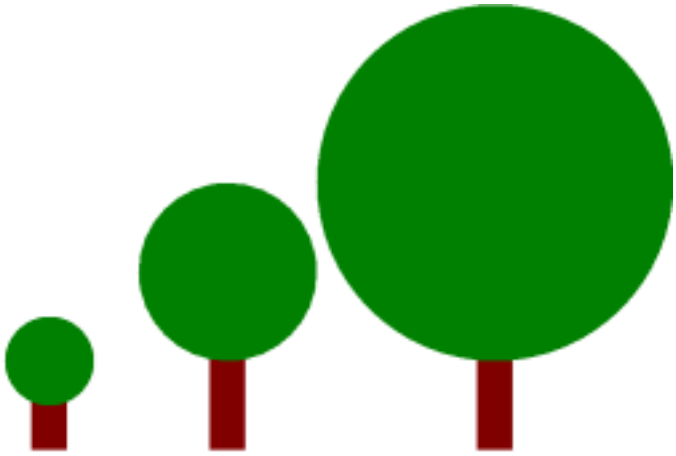
Der folgende Programmcode zeichnet einen Baum, der am Punkt (x, y) steht. Mit den entsprechenden Variablen können Dicke und Höhe des Stamms und der Radius der Krone:

```
window.setColor(Color.MAROON);
window.fillRectangle(x - dicke / 2, y - hoehe, dicke, hoehe);
window.setColor(Color.GREEN);
window.fillCircle(x, y - hoehe, radius);
```



Die Skizze zeigt auf, welche Bedeutung die verschiedenen Variablen haben.

Verwenden Sie den Codeausschnitt, um drei Bäume zu zeichnen. ***Der Codeausschnitt selbst soll unverändert bleiben.*** Die Dicke des Stamms soll immer 40 Pixel sein. Der erste Baum soll einen Radius von 50 und eine Stammhöhe von 100 haben. Der Radius der weitern Bäume soll jeweils doppelt so gross sein, die Höhe des Stamms jeweils um 100 Pixel höher.



Zusatzaufgabe 2.3

Die Methode `Jeda.randomInt()` liefert eine Zufallszahl zurück. Zum Beispiel liegt nach dieser Anweisung der Wert von x zwischen 0 und 99:

```
int x = Jeda.randomInt(100);
```

Verwenden Sie diese Methode und den Code aus Aufgabe 2.2, um einen Wald zu zeichnen.

Zusatzaufgabe 2.4

Wie gehen Sie vor, um die Werte zweier Variablen austauschen?

Schreiben Sie ein Programm, welches folgende Anweisungen verwendet, um zwei unterschiedliche Rechtecke zu zeichnen:

```
window.fillRectangle(x, y, width, height);
// Hier Inhalt von width und height tauschen!
window.fillRectangle(x, y, width, height);
```

Beim zweiten Rechteck sollen Breite und Höhe vertauscht sein. Die Zeichenbefehle dürfen aber nicht abgeändert werden!

2.3 Zeichenketten

Als `String` bezeichnet man eine beliebige Zeichenkette, darin können Buchstaben, Sonderzeichen, aber auch Ziffern enthalten sein. Strings werden immer in Anführungszeichen geschrieben.

2.3.1 Datentyp String

Auch mit Zeichenkette kann man „rechnen“, man kann sie nämlich z.B. aneinander reihen, dazu wird das Pluszeichen verwendet:

```
String name = "Siegenthaler";
String satz1 = "Mein Name ist " + name;
```

Man kann aber auch Zahlen in einen String integrieren:

```
int alter = 25;
String satz2 = "Ich bin " + alter + " Jahre alt.";
```

Da die Anweisung stets von links nach rechts ausgewertet wird, ist Vorsicht geboten, wenn man gleichzeitig rechnet und Strings aneinander reiht:

```
int a = 5;
int b = 7;
String satz3 = "a plus b ist gleich " + a + b;
```

Nun lautet der `satz3` wie folgt: "`a plus b ist gleich 57`". Zuerst wurde nämlich der Wert von `a` an den String angehängt, was einen neuen String ergab. Zu diesem wurde dann noch der Wert von `b` hinzugefügt. Korrekt hätte man schreiben müssen:

```
String satz4 = "a plus b ist gleich " + (a + b);
```

Eine Zeichenkette kann auch auf das Fenster ausgegeben werden:

```
window.drawString(10, 10, "Hallo");
```

Aufgabe 2.5

Schreiben Sie ein Programm, das dem Benutzer eine Rechnungsaufgabe stellt. Die verwendeten Zahlen sollen jedes Mal zufällig gewählt werden. Damit der Benutzer sein Resultat eingeben kann, rufen Sie folgenden Befehl auf:

```
int resultat = Dialog.readInt("Lösung:");
```

Geben Sie anschliessend die korrekte Lösung im Fenster aus.

2.4 Fliesskommazahlen in Java

2.4.1 Datentypen

Um Fliesskommazahlen zu speichern, stellt Java 2 Datentypen zur Verfügung: `float` und `double`. `float` belegt 32 bit im Speicher, `double` 64 bit. Wir werden in diesem Kurs ausschliesslich `double` verwenden.

2.4.2 Fliesskommazahlen

Auch Fliesskommazahlen kann man mit den üblichen mathematischen Operationen verknüpfen.

Zu beachten ist, dass sämtliche Operationen eine Fliesskommazahl als Resultat liefern, sobald mindestens a oder b eine Fliesskommazahl ist. Sind hingegen beides ganze Zahlen, so ist auch das Resultat ganzzahlig, was in insbesondere bei der Division häufig übersehen wird (siehe oben).

Operation	Mathematisch	Java
Addition	$a + b$	`a + b`
Subtraktion	$a - b$	`a - b`
Multiplikation	$a \cdot b$	`a * b`
Reelle Division	$a \div b$	`a / b`
Potenz	a^b	`Math.pow(a, b)`
Quadratwurzel	$\sqrt{a}$	`Math.sqrt(a)`
Betrag	$\|a\|$	`Math.abs(a)`

Beispiele:

```
double radius = 5.3;
double umfang = 2 * radius * Math.PI;

double a = 3.2;
double b = 7.8;
double c = Math.sqrt(Math.pow(a, 2) + Math.pow(b, 2));  // Pythagoras
```

Aufgabe 2.6

Kopieren Sie das Programm aus Aufgabe 2.5 und passen Sie es so an, dass die Rechnung Fliesskommazahlen enthält.

Überlegen Sie sich dabei, wie man aus einer ganzzahligen Zufallszahl eine Fliesskommazahl berechnen könnte. Beachten Sie dabei, dass die Ganzzahldivision ausgeführt wird, wenn Divisor und Dividend beide ganzzahlig sind!

Zusatzaufgabe 2.7

Schreiben Sie ein Programm, das quadratische Gleichungen der Form $a \cdot x^2 + b \cdot x + c = 0$ löst. Der Benutzer soll aufgefordert werden, die Fliesskomma-Werte für a, b und c einzugeben. Anschliessend werden die beiden Lösungen im Fenster ausgegeben.

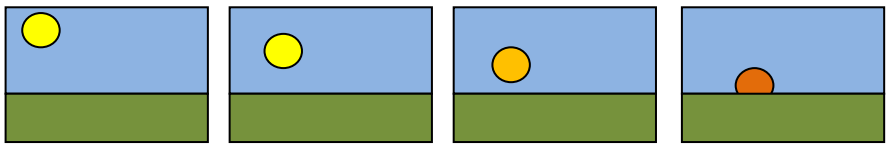
3 Animation

- → Sie wissen, wie in Jeda eine Animation programmiert wird.
- → Sie kennen den Unterschied zwischen lokalen Variablen und Instanzvariablen.
- → Sie verstehen, was mit Lebensdauer und Sichtbarkeit von Variablen gemeint ist.
- → Sie können aus RGB-Werten eine gewünschte Farbe "mischen".

3.1 Grundgerüst für Animationen

3.1.1 Aufbau einer Animation

In diesem Kapitel lernen wir, einfache Animationen zu programmieren. Im Gegensatz zu den bisherigen Programmen, welche einmal eine Zeichnung anfertigen, muss ein Animationsprogramm wiederholt veränderte Zeichnungen erstellen. Um beispielsweise einen Sonnenuntergang zu animieren, müssen nacheinander folgende Zeichnungen erstellt werden:



Einige Anweisungen müssen also wiederholt ausgeführt werden, dies nennen wir einen ***Animationsschritt***:

- Zeichnen des Himmels
- Zeichnen der Sonne
- Zeichnen der Erde
- Änderung der Sonnenposition
- Änderung der Sonnenfarbe

Allerdings gibt es auch einige Anweisungen, die nur einmal beim Start des Programms ausgeführt werden dürfen. Solche Anweisungen werden zusammengefasst ***Initialisierung*** genannt:

- Das Fenster erstellen.
- Die Startposition der Sonne festlegen.
- Die Startfarbe der Sonne festlegen.

3.1.2 Simulation

Für ein Animationsprogramm verwenden wir deshalb ein anderes Gerüst als für die bisherigen Programme:

```
public class Sunset extends Simulation {

    protected void init() {
       // Wird beim Start einmal aufgerufen.
    }
    protected void step() {
       // Wird 30 mal pro Sekunde aufgerufen.
    }
}
```

Dabei ist hier `Program` durch `Simulation` ersetzt worden. Anstatt der Methode `run()` gibt es die Methoden `init()` und `step()`. In diesen zwei Methoden können die Anweisungen für die Initialisierung und den Animationsschritt geschrieben werden:

- `init()`: Beim Start einer Simulation werden erst die Anweisungen in dieser Methode ausgeführt. Hier gehört also die Initialisierung hin.
- `step()`: Anschliessend werden die Anweisungen in der Methode `step()` 30 Mal pro Sekunde ausgeführt. Hier kann also der Animationsschritt programmiert werden.

Aufgabe 3.1

a) Erstellen Sie eine Jeda-Simulation-Klasse mit dem Namen `Sunset`.
b) Schreiben Sie in der Methode `init()` die Befehle, um ein neues Fenster zu erstellen und einen ausgefüllten Kreis in das Fenster zu zeichnen.
c) Verschieben Sie den Befehl für das Zeichnen des Kreises in die Methode `step()`. Was passiert?

3.2 Sichtbarkeit und Lebensdauer von Variablen

3.2.1 Lokale Variablen und Instanzvariablen

Der Grund für den Fehler in Aufgabe 3.1 ist, dass die `Window`-Variable nur innerhalb der Methode `init()` ***sichtbar*** ist. In der Methode `step()` kann darauf nicht zugegriffen werden. Alle Variablen, welche innerhalb einer Methode definiert werden, haben diese Eigenschaft. Sie werden ***lokale Variable*** genannt. Als Lösung für den Fehler bietet sich an, die Variable direkt in der Klasse zu definieren. Eine solche ***Instanzvariable*** ist in allen Methoden sichtbar:

```
public class Sunset extends Simulation {

    Window wind;  // Instanzvariable, ist in init() und step() sichtbar

    protected void init() {
        // ...
    }
}
```

3.2.2 Sichtbarkeit von Variablen

Der Sichtbarkeitsbereich einer Variablen legt fest, wo im Quellcode auf diese Variable zugegriffen werden kann. Dabei gelten für lokale Variablen und Instanzvariablen ganz unterschiedliche Regeln:

- ***Lokale Variable*** sind vom Ort ihrer Deklaration bis zur nächsten schliessenden geschweiften Klammer sichtbar. Da lokale Variablen innerhalb einer Methode definiert werden also höchstens bis zum Ende der Methode.
- ***Instanzvariablen*** sind unabhängig vom Ort ihrer Deklaration in allen Methoden der Klasse sichtbar. Als Ausnahme kann eine Instanzvariable durch eine lokale Variable mit dem gleichen Namen ***verdeckt*** werden. Innerhalb des Sichtbarkeitsbereichs der lokalen Variablen ist somit die Instanzvariable nicht sichtbar.

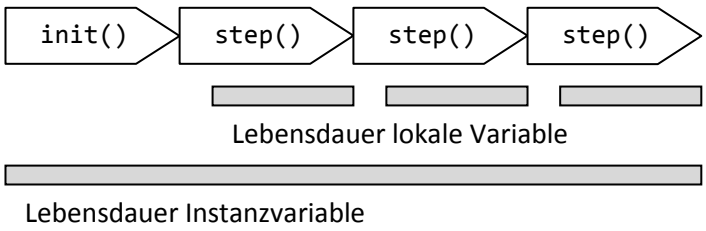
Das folgende Codebeispiel zeigt die Sichtbarkeitsregeln anhand zweier Variable mit dem Namen *x*:

```
public class Sunset extends Simulation {

    int x;  // Instanzvariable

    protected void init() {
        x = 2;  // Zugriff auf Instanzvariable
    }

    protected void step() {
        x = 4;  // Zugriff auf Instanzvariable
        int x;  // lokale Variable verdeckt Instanzvariable
        x = 2;  // Zugriff auf lokale Variable
    }  // Ende der Sichtbarkeit für lokale Variable
}
```

3.2.3 Lebensdauer

Neben der Sichtbarkeit ist die ***Lebensdauer*** ein wichtiges Merkmal von Variablen. Sie legt fest, wie lange eine Variable ihren Wert behält. Auch hier gibt es Unterschiede:

- Eine ***lokale Variable*** lebt nur während eines Methodenaufrufs. Beim nächsten Aufruf der Methode (zum Beispiel `step`) hat das Programm die Inhalte der lokalen Variablen "vergessen".
- Eine ***Instanzvariable*** lebt während allen Methodenaufrufen.



Die Lebensdauer der Variable darf aber nicht mit der Lebensdauer des Objekts, auf welche sie zeigt, verwechselt werden. Falls die bekannte Anweisung

```
Window wind = new Window();
```

in der Methode `step` einer Simulation steht, so wird bei jedem Durchlauf ein neues Fenster erzeugt. Beim Ende eines Aufrufs *vergisst* die Variable `wind`, dass sie auf ein Fenster zeigt, aber das Fenster selbst wird nicht geschlossen.

3.2.4 Zusammenfassung

Aus diesen Erörterungen über die Sichtbarkeit und Lebensdauer von Variablen lassen sich folgende Regeln ableiten:

- Verwenden Sie eine ***lokale Variable***, wenn deren Wert bei jedem Methodenaufruf neu berechnet wird.
- Verwenden Sie eine ***Instanzvariable***, wenn ein Wert in mehreren Methoden oder während aufeinanderfolgenden Aufrufen benötigt wird.

Die Eigenschaften dieser zwei Variablenarten können so zusammengefasst werden:

Eigenschaft	lokale Variable	Instanzvariable
Deklaration	innerhalb einer Methode	innerhalb einer Klasse
Sichtbarkeit (wo gültig?)	von Deklaration bis }	in allen Methoden, wenn nicht verdeckt
Lebensdauer (wann gültig?)	während aktuellem Methodenaufruf	während allen Methodenaufrufen

Aufgabe 3.2

a) Erweitern Sie ihre Simulation aus Aufgabe 3.1 so ab, dass sich der Kreis im Fenster bewegt. Definieren Sie dazu die `Window`-Variable sowie zwei Variable für die Koordinaten des Kreises als Instanzvariablen.

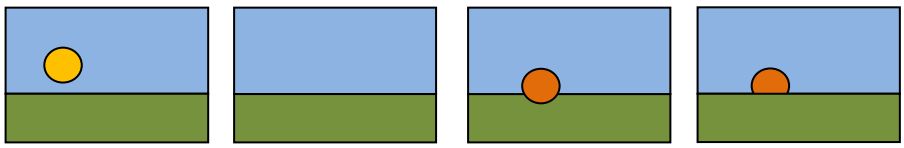
b) Vervollständigen Sie Ihre Animation zu einem Sonnenuntergang. Die Farbe der Sonne muss sich (noch) nicht ändern.

3.3 Double Buffering

3.3.1 Das Problem: Flackern

Sie haben wahrscheinlich festgestellt, dass Ihre Animation "flackert": Manchmal scheint die Sonne kurz zu verschwinden und nach ihrem Untergang scheint sie manchmal kurz durch die Erde hindurch.

Grund für dieses Flackern ist, dass durch die Abfolge der Zeichenbefehle kurze Zeitspannen entstehen, in welchen der Betrachter das unfertige Bild sieht:



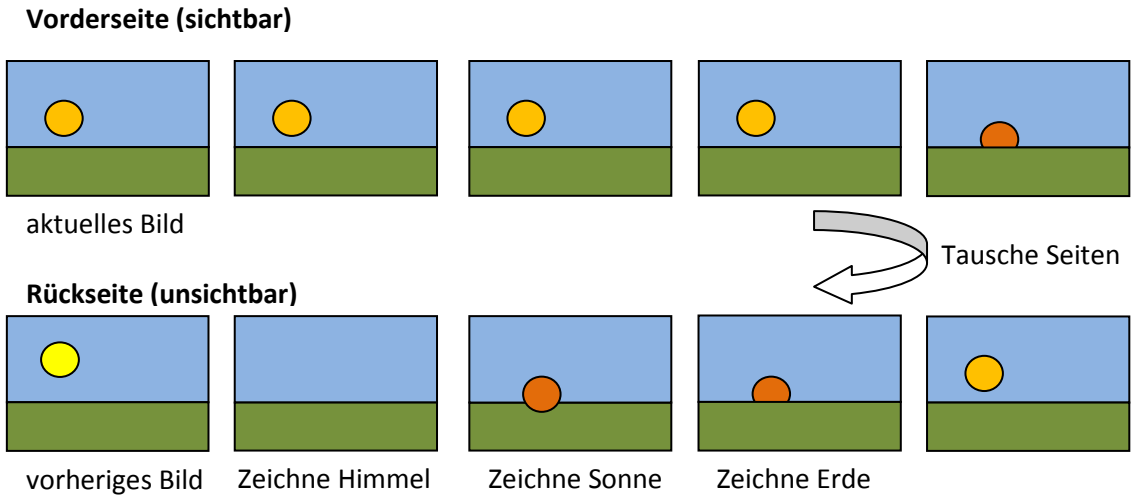
3.3.2 Die Lösung: Zwei Seiten

Um ein solches Flackern zu vermeiden, darf dem Betrachter folglich nur das fertig gezeichnete Bild gezeigt werden. Dazu benötigen wir zwei Zeichenflächen, die man sich anschaulich als Vorderseite und Rückseite eines Bildes vorstellen kann:

- Die ***Vorderseite*** wird dem Betrachter gezeigt, sie verändert sich nicht.

- Auf die ***Rückseite*** wird gezeichnet, sie bleibt jedoch unsichtbar.

Wenn das neue Bild auf der Rückseite fertigt gezeichnet worden ist, wird das Bild umgedreht, Vorder- und Rückseite werden also vertauscht. Nun sieht der Betrachter das neue Bild, das alte kann übermalt werden.



Diese Technik wird ***Double Buffering*** genannt. Das Jeda-Window unterstützt diese Technik. Durch die Anweisung

```
wind.setDoubleBuffered(b);
```

wird das Double Buffering eingeschaltet. Dabei kann für b einer der folgenden zwei Werte übergeben werden:

Wahrheitswert	Bedeutung
`true`	"ein", "ja", "wahr"
`false`	"aus", "nein", "falsch"

`true` und `false` heissen ***Wahrheitswerte***. Sie werden überall dort verwendet, wo es um eine Unterscheidung zwischen "an" und "aus", "ja" und "nein" oder "wahr" und "falsch" geht. Im nächsten Kapitel werden wir uns eingehender mit Wahrheitswerten beschäftigen.

Sobald das *Double Buffering* aktiviert ist, werden Zeichenbefehle auf der Rückseite des Fensters ausgeführt. Um das gezeichnete Bild betrachten zu können, müssen Vorder- und Rückseite mit der folgenden Anweisung getauscht werden:

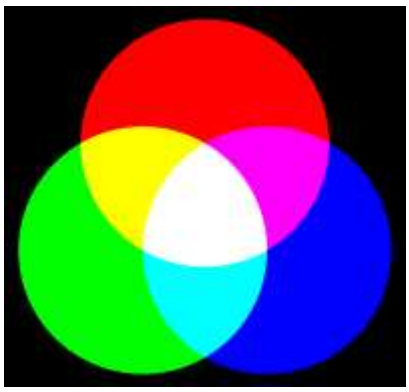
```
wind.flip();
```

Aufgabe 3.3

Aktivieren Sie das *Double Buffering* in Ihrer Simulation.

3.4 RGB-Farben

3.4.1 Additive Farbsysteme



Bei der additiven Farbmischung werden Faben durch die Überlagerung der drei Grundfarben Rot, Grün und Blau erzeugt. Das Bild veranschaulicht diesen Vorgang anhand von Scheinwerfern in den Grundfarben, deren Lichtkegel sich überschneiden.

Auf diese Weise werden auch die Farben beim Digitalprojektor oder Computerbildschirm erzeugt. Beim letzteren liegen einfach rote, grüne und blaue Bildpunkte eng nebeneinander. Die eigentliche Farbmischung findet bei additiven Farbsystemen im Auge und Gehirn statt.

3.4.2 RGB-Farben in Jeda

In Jeda können Sie beliebige RGB-Farben auf folgende Weise definieren:

`Color myColor = new Color(red, green, blue);`

Diese Anweisung erstellt eine neue Farbe mit den angegebenen Rot-, Grün und Blauwerten. Für `red`, `green` und `blue` können `int`-Werte zwischen 0 und 255 angegeben werden. Je grösser der Wert ist, desto heller ist der entsprechende Farbanteil.

`Color c2 = new Color(red, green, blue, alpha);`

Bei dieser Anweisung wird zusätzlich der sogenannte ***Alphawert*** angegeben. Er ist ein Mass für die Transparenz der Farbe. Auch hier werden Werte zwischen 0 und 255 akzeptiert. Je grösser der Wert ist, desto transparenter ist die Farbe. Wenn mit einer teiltransparenten Farbe gezeichnet wird, so scheinen die schon vorhandenen Formen durch.

So erstellte Farben können Sie anstelle der schon bekannten vordefinierten Farben verwenden:

```
wind.setColor(myColor);
```

Zusatzaufgabe 3.4

Erweitern Sie Ihre Sonnenuntergangs-Animation so, dass sich die Farbe der Sonne von Gelb nach Rot ändert.

4 Verzweigungen

- → Sie können erklären, wie Verzweigungen in Programmiersprachen aufgebaut sind.
- → Sie können eine umgangssprachliche Bedingung als Wenn-dann-sonst-Aussage formulieren und diese in ein Java-Programm umsetzen.
- → Sie können durch das Verknüpfen von Vergleichen und logischen Operationen Wahrheitswerte definieren.

4.1 Welcher Weg?

4.1.1 Motivation

Unsere Sonnenuntergangs-Simulation ist noch verbesserungsfähig: Zum Beispiel müsste sich das ganze Bild abdunkeln, wenn die Sonne untergegangen ist. Dazu müssen aber nun andere Anweisungen ausgeführt werden, der Ablauf des Programms muss sich ändern.

4.1.2 Verzweigungen

Beim Wandern treffen Sie immer wieder auf Wegweiser, welche eine Verzweigung markieren. Dann gilt es jeweils eine Entscheidung zu treffen: "Soll ich den Wanderweg nehmen oder den Bergwanderweg?"

Unsere Entscheidung hängt von verschiedenen ***Bedingungen*** ab: Sind wir müde? Tragen wir Bergschuhe? Wir können zum Beispiel sagen:

> ***Wenn*** wir Bergschuhe tragen und nicht müde sind, ***dann*** wählen wir den Bergweg, ***sonst*** wählen wir den Wanderweg.

In Programmiersprachen werden Verzweigungen ganz ähnlich formuliert. Sie haben folgende Form:

> ***Wenn*** (Bedingung), ***dann*** (Anweisung 1), ***sonst*** (Anweisung 2).

Beim Programmieren von Verzweigungen gibt es zwei wichtige Punkte zu beachten:

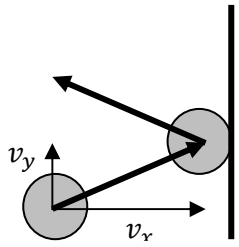
- Die Umsetzung einer alltagssprachlichen Formulierung in eine Wenn-dann-sonst-Anweisung, welche sich dann einfach in entsprechenden Quellcode übersetzen lässt.
- Die Formulierung der Bedingung mit Hilfe von ***Wahrheitswerten***.

Aufgabe 4.1
Formulieren Sie das einfache Glücksspiel "Münzwurf" mit einer Verzweigung.

Zusatzaufgabe 4.2

Eine Animation soll einen Ball darstellen, welcher sich innerhalb des Fensters bewegt und an den Rändern abprallt.

Formulieren Sie (auf Deutsch) bedingte Anweisungen, welche die Formulierung *„an den Rändern abprallt* umsetzen“. Sie kennen die aktuellen Koordinaten(x, y), die Bewegungsrichtung (v_x, v_y) und den Radius r des Balls sowie die Breite und Höhe des Fensters.



Tipp: Beim Abprallen ändert jeweils eine Komponente der Bewegungsrichtung das Vorzeichen.

4.2 Wahrheitswerte

4.2.1 Datentyp boolean

Im letzten Kapitel haben wir die Wahrheitswerte `true` und `false` kennengelernt. Bei bedingten Anweisungen werden Wahrheitswerte verwendet, um auszudrücken, ob eine Bedingung zutrifft oder nicht. Der Datentyp von Wahrheitswerten ist `boolean`[1], die zwei möglichen Werte sind `true` und `false`:

Wahrheitswert	Bedeutung
`true`	ein, ja, wahr
`false`	aus, nein, falsch

Wir können einen Wahrheitswert auf verschiedene Arten erhalten:

- durch Vergleichen von Zahlen,
- durch Kombination von bekannten Wahrheitswerten oder
- durch Aufruf von bestimmten Abfrage-Methoden.

4.2.2 Zahlen Vergleichen

Häufig kommen Wahrheitswerte durch den Vergleich von Zahlen zustande. Beachten Sie die Schreibweise der Vergleichsoperatoren in Java, insbesondere der Gleichheit.

In der folgenden Tabelle sind die Vergleichsoperationen für Zahlen zusammengefasst. Dabei sind a und b Zahlen, das Ergebnis eines Vergleichs ist ein Wahrheitswert:

Operation	Mathematik	Java
Ist gleich	$a = b$	`a == b`
Ist nicht gleich	$a \neq b$	`a != b`
Ist kleiner als	$a < b$	`a < b`
Ist kleiner als oder gleich	$a \leq b$	`a <= b`

[1] Namensgeber ist George Boole, 1815 - 1864, englischer Mathematiker, welcher mit seinem Logikkalkül die moderne mathematische Logik begründet und damit einen Grundstein für die Digitaltechnik gelegt hat.

Operation	Mathematik	Java
Ist grösser als	$a > b$	`a > b`
Ist grösser als oder gleich	$a \geq b$	`a >= b`

Ein häufiger Fehler beim Programmieren in Java ist das Verwechseln der Zuweisung = mit dem Gleichheitsoperator ==, welcher aus zwei Gleichheitszeichen besteht. Beachten Sie den Unterschied bei folgendem Beispiel:

```
Window w = new Window();
int a = Dialog.readInt("Bitte eine Zahl eingeben:");
boolean isZero = a == 0;
w.drawString(10, 10, "Zahl " + a + " ist Null: " + isZero);
```

Hier wird vom Benutzer eine Zahl eingegeben und der Variablen a zugewiesen. Anschliessend wird der Wert von a mit Null verglichen. Das Resultat dieses Vergleichs wird der Variable `isZero` zugewiesen. Schliesslich wird ein String mit dem Inhalt der Variablen a und `isZero` verknüpft und ausgegeben.

Aufgabe 4.3

Schreiben Sie ein Java-Programm, welches eine ganze Zahl einliest und folgende Eigenschaften der Zahl überprüft:

- Die Zahl ist grösser als 10.
- Die Zahl ist kleiner als 100.
- Die Zahl ist gleich 42.
- Die Zahl ist nicht gleich -5.

Zusatzaufgabe 4.4

a) Erweitern Sie das Programm aus Aufgabe 4.3 so, dass die Eigenschaft "a ist gerade" überprüft wird.
b) Was ist das Resultat des untenstehenden Vergleichs? Haben Sie eine Erklärung? Wie könnten Sie das Problem umgehen?

```
1 == (1.0 / 49) * 49
```

4.2.3 Wahrheitswerte kombinieren

Wahrheitswerte können mit sogenannten ***logischen Operationen*** kombiniert werden, welche unserer intuitiven Vorstellungen von Wahrheitswerten entsprechen:

- Die ***Und***-Operation zweier `boolean`-Werten liefert genau dann `true`, wenn beide Werte `true` sind.
- Die ***Oder***-Operation zweier `boolean`-Werten liefert `true`, wenn mindestens einer der beiden Werte `true` ist.

- Die ***Nicht***-Operation eines `boolean`-Wertes liefert `true`, wenn der ursprüngliche Wert `false` ist.

In der folgenden Tabelle sind die mathematische Schreibweise und die Java-Notation dieser Operationen dargestellt. Dabei stehen v und w für Wahrheitswerte:

Operation	Mathematik	Java
Logisches Und	$v \wedge w$	`v && w`
Logisches Oder	$a \vee b$	`v || w`
Logisches Nicht	$\neg v$	`!v`

Zum Beispiel möchten wir überprüfen, ob der Wert der int-Variable a einer Schulnote entspricht, also zwischen 1 und 6 liegt. Anders formuliert:

a ist grösser oder gleich 1 ***und*** a ist kleiner oder gleich 6.

Die Umsetzung in Java sieht folgendermassen aus[2]:

```
(a >= 1) && (a <= 6)
```

Zustäzlich kann durch Umdrehen des ersten Vergleichs eine verständlichere Darstellung erreicht werden:

```
(1 <= a) && (a <= 6)
```

Aufgabe 4.5

Schreiben Sie ein Java-Programm, welches eine ganze Zahl einliest und folgende Aussagen beantwortet:

- Die Zahl ist kleiner als -100 oder grösser als 100.
- Die Zahl ist nicht 2 und nicht 3 und nicht 4.

Zusatzaufgabe 4.6

a) Erweitern Sie das Programm aus Aufgabe 4.5 so, dass die Aussage "Die Zahl ist weder gerade noch negativ" überprüft wird.
b) Schreiben Sie die Überprüfung aus Teilaufgabe a) ohne den logischen und-Operator zu verwenden.
c) Stellen Sie jede Vergleichsoperation von Zahlen (Abschnitt 4.2.2) nur mit Hilfe der Vergleichsoperation "kleiner als" und den logischen Verknüpfungen dar.

[2] Die Klammern sind freiwillig, sie dienen des besseren Lesbarkeit.

4.3 Verzweigungen in Java

4.3.1 if-else-Anweisung

Nun betrachten wir, wie Verzweigungen in Java programmiert werden können. Dazu benötigen wir

- eine Bedingung ***B***, die einen Wahrheitswert ergibt.
- Anweisungen ***T***, welche ausgeführt werden, wenn der Wahrheitswert wahr (`true`) ist.
- Anweisungen ***F***, die ausgeführt werden, falls der Wahrheitswert falsch (`false`) ist.

In Java werden für die Verzweigung die Worte `if` und `else` verwendet, deshalb wird auch von einer `if`-Anweisung oder `if-else`-Anweisung gesprochen. Sie wird folgendermassen geschrieben:

```
if (B) {
    T
}
else {
    F
}
```

Dabei müssen natürlich ***B***, ***T*** und ***F*** durch entsprechende Programmkonstrukte ersetzt werden, zum Beispiel:

```
if (Jeda.randomInt(2) == 0) {
    wind.drawString(10, 10, "Kopf");
}
else {
    wind.drawString(10, 10, "Zahl");
}
wind.drawString(100, 10, "gewinnt.");
```

Diese Anweisung ist die Java-Umsetzung eines Münzenwurfs. Die geschweiften Klammern begrenzen dabei die bedingten Anweisungen, welche nur ausgeführt werden sollen, wenn die Bedingung zutrifft[3]. Nach der schliessenden geschweiften Klammern folgen wieder Anweisungen, welche in jedem Fall ausgeführt werden.

Manchmal bleibt der F-Teil einer Verzweigung auch leer, er wird nicht benötigt. In diesem Fall darf man ihn in Java auch gleich ganz weglassen:

```
int a = Dialog.readInt("Bitte eine ganze Zahl eingeben:");
if (a > 0) {
    double b = Math.sqrt(a);
    wind.drawString(10, 30, "Die Wurzel von a ist " + b);
}
wind.drawString(10, 10, "a = " + a);
```

Im obenstehenden Beispiel werden zwei Anweisungen nur bedingt ausgeführt. Die erste und letzte Anweisung werden in jedem Fall ausgeführt. Wie wir im letzten Kapitel gelernt haben, ist die Variable b nur bis zur schliessenden geschweiften Klammer gültig.

[3] Wenn nur eine bedingte Anweisung vorhanden ist, können die geschweiften Klammern weggelassen werden. Dies kann jedoch zu missverständlichem Code führen und wird nicht empfohlen.

Aufgabe 4.7

Programmieren Sie die Animation eines Balles, welcher sich nach oben bewegt, bis er den oberen Fensterrand erreicht.

Hinweis: Formulieren Sie die Verzweigung als Wenn-dann-sonst-Aussage, ***bevor*** Sie mit dem Programmieren beginnen.

Aufgabe 4.8

Programmieren Sie die Animation eines Balles, welcher sich innerhalb des Fensters bewegt und an den Rändern abprallt.

Hinweis: Die Breite und Höhe des Fensters `wind` können mit den Methodenaufrufen `wind.getWidth()` und `wind.getHeight()` abgefragt werden.

4.3.2 Mehrfachverzweigungen

Müssen mehr als zwei Fälle unterschieden werden, passiert dies oft fälschlicherweise wie folgt:

```
if (a < 0) {
    A
}
if (a == 0) {
    B
}
else {
    C
}
```

Aufgabe 4.9

Welche der Teile A, B und C werden im oben stehenden Beispiel ausgeführt, wenn

a) a gleich 0 ist?
b) a grösser als 0 ist?
c) a kleiner als 0 ist?

Schreibt man das Beispiel korrekt hin, so ergibt sich folgender Code:

```
if (a < 0) {
    A
}
else {
    if (a == 0) {
        B
    }
    else {
        C
    }
}
```

Stellen wir uns ein Beispiel mit 5 oder 10 verschiedenen Unterscheidungen vor, so wird der else-Teil immer weiter eingerückt, so dass unser Programm völlig unübersichtlich wird. Aus diesem Grund wenden wir – aber nur in Falle einer Mehrfachunterscheidung – die Regel an, dass man geschweifte Klammen weglassen darf, sofern nur eine einzige Anweisung drin steht. Im Beispiel oben sieht man grau unterlegt, dass im äusseren (nicht unterlegten) else-Teil nur eine Anweisung steht, nämlich das grau unterlegte if mit dazugehörigem else-Teil.

Wir dürfen also die geschweiften Klammern um den grau unterlegten Block weglassen und den Code neu wie folgt darstellen:

```
if (a < 0) {
    A
}
else if (a == 0) {
    B
}
else {
    C
}
```

Nun stehen die drei Anweisungsblöcke A, B und C alle auf derselben Ebene und unser Konstrukt besteht aus

- mindestens einem if-Teil
- keinem oder beliebig vielen else if-Teilen
- höchstens einem else-Teil

Beachten Sie, dass der ***else-Teil*** (also der Teil ohne if) ***nie eine Bedingung*** enthält!

Achten Sie beim Programmieren darauf, dass Sie ***vor und nach einer if-else-Anweisung*** jeweils ***eine Leerzeile einfügen***, um Missverständnisse zu vermeiden.

Aufgabe 4.10

Verwenden Sie als Grundlage für diese Aufgabe die Animation aus Aufgabe 4.8. Erweitern Sie die step-Methode so, dass der Ball je nach Richtung eine andere Farbe hat:

a) Grün, wenn er sich nach oben links bewegt,
b) Rot, wenn er sich nach unten links bewegt,
c) Blau, wenn er sich nach oben rechts bewegt,
d) Gelb, wenn er sich nach unten rechts bewegt.

5 Schleifen

5.1 Wiederholungen

5.1.1 Motivation

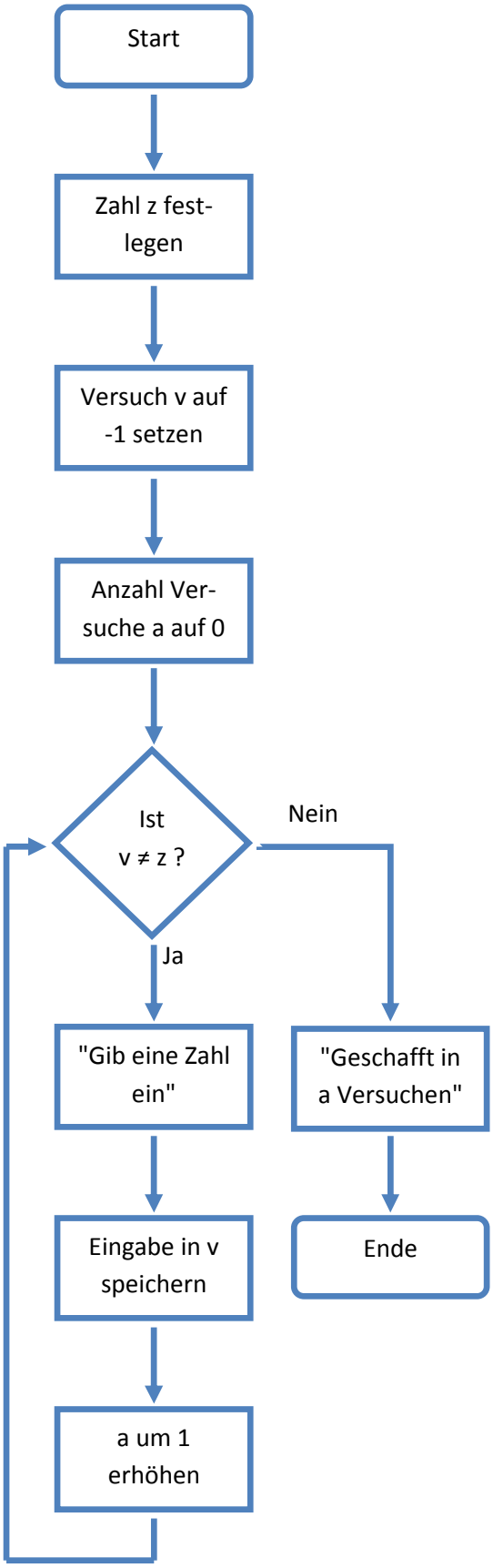
Beim Programmieren sollte man vermeiden, denselben Codeblock mehrmals zu schreiben. Erstens wird so der Quelltext unnötig lang, andererseits wissen Sie aus eigener Erfahrung, wie lästig Copy&Paste-Fehler sind. Oder man stelle sich nur vor, eine Anweisung müsse 1‘000 Mal oder 1 Million Mal ausgeführt werden…

Zudem ist Code, der mehrfach vorhanden ist, schwer zu warten. Wenn Sie Ihr Programm in einem halben Jahr überarbeiten müssen, müssen Sie unter Umständen daran denken, den fehlerhaften Code an mehreren Stellen zu korrigieren und nicht nur an einer.

Eine Programmiersprache bietet deshalb die Möglichkeit, Codestellen zu markieren, die wiederholt werden sollen. Sie kennen solche Markierungen aus der Musik, dort gibt es Wiederholungszeichen, damit dieselbe Passage nicht mehrfach notiert werden muss.

5.1.2 Wiederholungen in der Programmierung

In der Programmierung sind die Möglichkeiten natürlich vielfältiger als in der Musik. Einerseits können wir angeben, dass eine Wiederholung stattfinden soll, bis eine Bedingung erfüllt ist, z.B. so lange nach eine Zahl fragen, bis der/die BenutzerIn sie richtig erraten hat. Andererseits muss in der Wiederholung nicht zwangsläufig bei jedem Durchgang exakt dasselbe geschehen. Im Beispiel *Zahlen raten* könnte man beispielsweise jeweils ausgeben, wie viele Versuche bereits erfolgt sind. Der Vorgang bleibt stets derselbe (dem/der BenutzerIn anzeigen, was er/sie tun soll, die Eingabe entgegen nehmen und anschliessend verarbeiten). Nur läuft der Vorgang jeweils mit verschiedenen Werten ab. Das System weiss, wie oft bereits geraten wurde und erhöht den Wert bei jedem erneuten Versuch. Der/die BenutzerIn gibt ja auch nicht jedes Mal dieselbe Zahl ein, also muss zwar eine Zahl überprüft werden, aber nicht in jeder Wiederholung dieselbe.



5.2 Schleifen in Java

5.2.1 Die while-Schleife

In Java gibt es verschiedene Konstrukte (Schleifen genannt), um Wiederholungen im Quellcode zu markieren. Die einfachste Schleife ist die while-Schleife. Sie sieht folgendermassen aus:

```
while (B) {
    A
}
```

Solange die Bedingung B erfüllt (also wahr) ist, werden die in den nachfolgenden, geschweiften Klammern aufgelisteten Anweisungen A ausgeführt.

Aufgabe 5.1

Schreiben Sie ein Java-Programm (Jeda Program Class), welche das oben erwähnte Zahlenratespiel realisiert. Gehen Sie exakt wie im Flussdiagramm oben beschrieben vor. Verwenden Sie aber aussagekräftigere Variablennamen als im Flussdiagramm (also z.B. zahl statt z und versuch statt v, etc.).

Als Erweiterung könnten Sie bei jedem Rateversuch noch ausgeben, wie oft schon geraten wurde. Programmieren Sie diesen Schritt aber erst, wenn Ihr Spiel korrekt funktioniert.

5.2.2 Verschachtelte Anweisungen

Schleifen, aber auch Verzweigungen führen bei erfüllter Bedingung beliebige Anweisungen aus. Es kann sein, dass in einer while-Schleife nur eine einzige Anweisung steht, es können aber auch ganz viele sein. In unserem Zahlen-Ratespiel wurde in der Schleife zuerst eine Meldung ausgegeben, dann nach einer Zahl gefragt und schliesslich die Anzahl Versuche hochgezählt. Aber auch eine if-else-Anweisung ist eine gültige Anweisung, die innerhalb einer Schleife vorkommen darf.

```
while (A) {
    if (B) {
        M
    }
    else {
        N
    }
}
```

So können wir in einer Schleife bei jedem Durchgang eine Bedingung B prüfen und gezielt reagieren. Unsere Programme werden so vielseitiger und flexibler.

Aufgabe 5.2

Wenn wir eine if-else-Anweisung in der Schleife einsetzen, können wir unser Ratespiel entscheidend verbessern:

Verändern Sie Ihr Ratespiel so, dass nach jeder Eingabe ausgegeben wird, ob die zu erratende Zahl grösser oder kleiner ist, als der eingegebene Versuch.

5.2.3 Endlosschleifen

Damit unsere Programme korrekt funktionieren und der Computer nicht abstürzt, ist es entscheidend, dass jede Schleife einmal verlassen wird. Wenn unser Programm ungünstig aufgebaut ist, könnte sich eine Endlosschleife ergeben, was den Computer vollständig auslasten würde.

Im oben erwähnten Beispiel kann das zwar auch geschehen, wenn der/die BenutzerIn die richtige Zahl nie erraten kann. Der Computer wartet aber stets auf eine Benutzereingabe und ist somit nicht ununterbrochen beschäftig.

Schwieriger hingegen wird es, wenn ein automatisierter Prozess abläuft. Es muss dafür gesorgt werden, dass das Programm Schritte unternimmt, die dafür sorgen, dass die Bedingung nicht mehr wahr ist.

```
Window window = new Window();
int x = 1;
while (x <= 10) {
    window.drawString(10, 20*x, "Schleifendurchgang " + x);
    x = x + 1;
}
```

Aufgabe 5.3

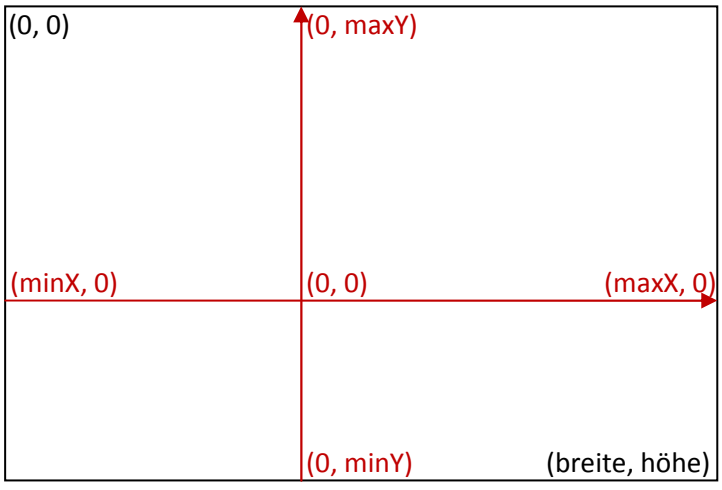
Erstellen Sie eine neue Jeda-Klasse und kopieren Sie den obenstehenden Code. Lassen Sie das Programm so laufen und schauen Sie sich das Resultat an.

Nun löschen Sie die Zeile, die x in jedem Schleifendurchgang vergrössert. Was stellen Sie fest? Öffnen Sie auch den Tastmanager (Ctrl+Alt+Del drücken, dann Taskmanager anklicken) und beobachten Sie, wie viele Systemressourcen Ihr Programm konsumiert.

5.3 Graphen von Funktionen zeichnen

5.3.1 Ein mathematisches Koordinatensystem

Neben dem bekannten Fenster bietet das Jeda-Framework auch noch ein Fenster, das speziell für die Anzeige von Graphen von Funktionen geeignet ist (genannt `GraphWindow`). Es erlaubt nämlich, ein eigenes Koordinatensystem in diesem Fenster zu definieren.



Das schwarz dargestellte Koordinatensystem kennen Sie bereits, es bezeichnet die Fenstergrösse in Pixel. Das rot dargestellte Koordinatensystem kann beliebig im Fenster platziert werden,

indem man die vier Werte (minX, minY, maxX, maxY) angibt. Dieses `GraphWindow` erlaubt auch die Angabe von `double`-Werten (im Gegensatz zum normalen `Window`).

```
GraphWindow graph = new GraphWindow(800, 600);
graph.setBounds(-10.0, -2.0, 10.0, 2.0);
```

Der erste Befehl erzeugt ein GraphWindow, das in der Breite 800, in der Höhe 600 Pixel aufweist. Mit dem zweiten Befehl definiert man das mathematische Koordinatensystem. Der im Fenster sichtbare Teil der X-Achse beginnt folglich bei -10 und endet bei 10, der sichtbare Y-Achsenausschnitt geht von -2 bis 2.

Das GraphWindow bietet noch die zwei weitere, wichtige Befehle `graph.moveTo(x, y)` und `graph.lineTo(x, y)`. Der erste Befehl bewegt den Zeichnungsstift zur angegebenen Position, dabei wird nichts gezeichnet. Beim zweiten Befehl wird von der aktuellen Position des Zeichnungsstiftes zur angegebenen Position eine Linie gezeichnet. Durch diese beiden folgenden Anweisungen wird eine Linie vom Ursprung zum Punkt (1, 1) gezeichnet.

```
graph.moveTo(0, 0);
graph.lineTo(1, 1);
```

Aufgabe 5.4

Öffnen Sie in einer neuen Jeda Program Class ein GraphWindow. Definieren Sie ein Koordinatensystem mit einem X- und Y-Achsenausschnitt von je -5 bis 5. Zeichnen Sie darin die beiden Strecken A: (-1,-1)→(1, 1) und B: (-2,2)→(4, 2).

Selbstverständlich kann wie gewohnt eine Farbe gesetzt werden mittels `graph.setColor(Color.RED)`. Zudem bietet das GraphWindow die Möglichkeit, die Koordinatenachsen einzublenden mit Hilfe der beiden Befehle `graph.drawXAxis()` und `graph.drawYAxis()`.

5.3.2 Wie zeichnet man Graphen von Funktionen?

Wollen wir Graphen von mathematischen Funktionen zeichnen, so stellen wir uns vor, wir würden entlang der X-Achse gehen und bei jedem Schritt auf der X-Achse die dazugehörige Y-Koordinate ausrechnen und dann den entsprechenden Punkt (x, y) zeichnen. Dazu brauchen wir also wieder eine Schleife, in der folgendes geschieht:

- Neue Position auf der X-Achse ausrechnen (X-Koordinate x verändern)
- Den Funktionswert von x berechnen (ergibt die Y-Koordinate y)
- Von der vorherigen Position zur neu errechneten Position (x, y) eine Linie zeichnen

Gestartet wird auf der linken Seite des Fensters (also mit minX als x-Wert). Der ganze Vorgang wird solange wiederholt, bis der x-Wert den rechten Rand des Fensters erreicht.

Aufgabe 5.5

Öffnen Sie in einer neuen Jeda Program Class ein GraphWindow. Definieren Sie ein Koordinatensystem wie in der vorherigen Aufgabe. Lassen Sie die Koordinatenachsen in schwarz anzeigen. Zeichnen Sie dann in rot die Funktion f(x) = 2*x.

Aufgabe 5.6

Erstellen Sie ein GraphWindow in einer Grösse, die zu der nachfolgenden Aufgabe passt. Definieren Sie ein sinnvolles Koordinatensystem zur Anzeige des Graphen der Funktion $f(x) = x^2$. Lassen Sie die Koordinatenachsen wiederum in schwarz anzeigen. Zeichnen Sie dann in rot die Funktion. Die Funktion soll so dargestellt werden, dass die interessante Stelle ihres Graphen gut sichtbar ist. Zudem sollen die Einheiten auf beiden Achsen gleich gross sein!

Zusatzaufgabe 5.7

Lassen Sie in einem GraphWindow sinnvoller Grösse mit passend gewähltem Koordinatensystem die beiden Funktionen $f(x) = \sin(x)$ und $g(x) = -2x^2 - x + 1$ in verschiedenen Farben zeichnen.

Schreiben Sie zudem die beiden Graphen mit Hilfe der Methode graph.drawString(s) an.

Zusatzaufgabe 5.8

Wir möchten untersuchen, wie sich der Parameter a in der Funktion $f(x) = a*x^2$ auf ihren Graphen auswirkt. Animieren Sie in einem GraphWindow sinnvoller Grösse mit passend gewähltem Koordinatensystem die Funktion $f(x) = a*x^2$ wie folgt:

In einer neuen *JedaSimulationClass* soll in jedem Simulationsschritt der Graph der Funktion f neu gezeichnet werden – basierend auf einem neuen a-Wert. Pro Simulationsschritt wird also ein kompletter Graph gezeichnet, in jedem Simulationsschritt verändert sich aber der a-Wert.

6 Interaktive Animation

- → Sie können eine interaktive Animation programmieren.
- → Sie wissen, wie und warum Konstanten in Java definiert werden.
- → Sie verstehen die Übergabe und Rückgabe von Werten an bzw. von Methoden.

6.1 Konstanten

Bestimmte Werte und Objekte, welche in einem Programm häufig benötigt werden, können als Konstanten definiert werden. Sie haben bereits Farb-Konstanten wie `Color.WHITE` verwendet. Konstanten sind nichts anderes als Variablen, welche nicht verändert werden können.

6.1.1 Unveränderbarkeit

Im Gegensatz zu "normalen" Variablen, welchen jederzeit ein neuer Wert zugeordnet werden kann, kann der Wert einer Konstante nie verändert werden. Dadurch können viele Probleme beim Programmieren verhindert werden.

```
w.setColor(Color.WHITE);        // Ok
w.fillCircle(100, 100, 50);
Color.WHITE = Color.BLACK;      // Fehler
```

Auf der dritten Zeile wird der Java-Compiler eine Fehlermeldung ausgeben. Das ist gut so, sonst würde Color.WHITE plötzlich den Farbwert für Schwarz enthalten!

6.1.2 Definition von Konstanten

Konstanten werden, wie auch Instanzvariablen, direkt innerhalb der Klasse definiert. Um eine Konstante zu erhalten, wird der Instanzvariable das Schlüsselwort `final` vorangestellt:

```
public class MyProgram extends Simulation {

  int x;
  int y;
  final int RADIUS = 30;

}
```

In diesem Beispiel ist sichergestellt, dass die Konstante `RADIUS` nicht mehr verändert werden kann. Die spätere Zuweisung eines neuen Wertes zu einer solchen Konstante ist verboten. Üblicherweise werden in Java die ***Namen von Konstanten*** immer mit ***Grossbuchstaben*** geschrieben.

6.1.3 Verwendung von Konstantendefinitionen

Sie sollten möglichst oft Konstanten verwenden, sie erleichtern das Programmieren zum Beispiel in folgenden Fällen:

- Definieren Sie jede Variable, welche nicht verändert werden soll, als Konstante und schreiben deren Namen komplett mit Grossbuchstaben. So können Sie veränderliche und unveränderliche Werte sofort unterscheiden. Und Java überprüft für Sie automatisch, dass Konstanten wirklich nicht verändert werden.

- Definieren Sie jede Zahl, welche mehr als einmal vorkommt, als Konstante, zum Beispiel wenn Sie mehrere Kreise mit dem gleichen Radius zeichnen wollen. Wenn Sie eine Konstante verwenden, müssen Sie den Wert nur an einer Stelle ändern.
- Geben Sie den Konstanten einen sinnvollen Namen, so ist Ihr Code einfacher zu verstehen. Zum Beispiel könnten Sie eine Konstante `EINZUG_LINKS` verwenden, damit Ihre Ausgabe im Fenster stets denselben Abstand vom linken Rand hat. Dieser Wert ist viel besser zu verstehen als die nackte Zahl `20`. Zudem können Sie so jederzeit schreiben `2*EINZUG_LINKS`, was viel besser nachvollziehbar ist, als wenn einfach plötzlich die Zahl `40` dasteht.

Aufgabe 6.1

Vor welche der Variablen kann `final` gesetzt werden, so dass der Code weiterhin korrekt ist?

```
int a = 2;
int b = 3;

if (b == a) {
  a = b;
}
```

Aufgabe 6.2

Verwenden Sie als Grundlage für diese Aufgabe die Animation mit dem abprallenden Ball aus Aufgabe 4.5.

Ändern Sie Ihren Programmcode so ab, dass in der Methode `step()` keine Zahlen vorkommen. Verwenden Sie stattdessen Konstanten.

6.2 Methoden

6.2.1 Methodenkopf/Signatur

Wir haben schon Anweisungen wie die folgende kennengelernt:

```
wind.drawCircle(100, 60, 50);
```

Hier wird dem Fenster, auf welches die Variable `wind` verweist, der Befehl zum Zeichnen eines Kreises geschickt. Wir sprechen dabei auch von einem ***Methodenaufruf***. Eine Methode ist ein Konstrukt, das mehrere Java-Anweisungen zusammenfasst und unter einem Namen (hier `drawCircle`) verfügbar macht. In anderen Programmiersprachen wird dieses Konstrukt *Funktion* oder *Prozedur* geannt, in Java spricht man von einer Methode. Damit eine Methode aufgerufen werden kann, muss sie definiert sein. Der ***Methodenkopf*** (auch ***Signatur*** genannt) für `drawCircle()` sieht folgendermassen aus:

```
void drawCircle(int x, int y, int radius)
```

Dabei definiert der Methodenkopf, wie genau die Methoden verwendet werden kann. Aus diesem Beispiel kann abgelesen werden, dass

- die Methode ***keinen*** Wert zurückliefert (`void`),
- der Methodenname `drawCircle` lautet,
- die Methode drei Werte übergeben werden müssen (`x`, `y` und `radius`),
- der Methode an Stelle von `x`, `y` und `radius` je einen `int`-Wert übergeben werden muss.

Allgemein stellt der ***Methodenkopf*** eine ***Vorlage für einen Befehl*** an ein bestimmtes Objekt dar. Der Methodenkopf besteht aus

- Rückgabetyp
- Name der Methode
- Parameterliste

Nach dem Methodenkopf folgt in geschweiften Klammern der Körper der Methode. Dort werden die Java-Anweisungen aufgeführt, die ausgeführt werden, wenn die Methode aufgerufen wird.

Im Folgenden betrachten wir die Bedeutung des Rückgabetyps und der Parameterliste im Detail.

6.2.2 Parameterliste

Manche Methoden benötigen zusätzliche Informationen, um ihre Aufgabe erfüllen zu können. Zum Beispiel muss die Methode `drawCircle()` wissen, wie gross der Kreis sein soll. Dies wird durch einen ***Parameter*** ausgedrückt. Ein Parameter setzt sich aus einem Datentyp und einem Namen zusammen.

Dadurch weiss jemand, der die Methode verwendet, was für ein Wert übergeben muss und wozu der Wert benötigt wird. Mehrere Parameter werden durch Kommas getrennt.

```
void drawCircle(int x, int y, int radius)
```

Die Reihenfolge der Parameter ist dabei sehr wichtig, sie muss bei den Methodenaufrufen berücksichtigt werden.

6.2.3 Rückgabetyp

Der Rückgabetyp steht zuvorderst beim Methodenkopf. Er definiert, was für einen Wert die Methode zurückliefert. Eine spezielle Bedeutung hat dabei das Wort `void`, welches festlegt, dass eben kein Wert zurückgeliefert wird, wie in der Methode `drawCircle()` der Klasse `Window`:

```
void drawCircle(int x, int y, int radius)
```

Andere Methoden liefern einen Wert zurück, wie etwa `randomInt()` der Klasse `Jeda`. Der Methodenkopf dieser Methode sieht so aus:

```
int randomInt(int max)
```

Hier kann abgelesen werden, dass die Methode einen `int`-Wert zurückliefert. In diesem Fall kann dann der zurückgelieferte Wert in einer Variable gespeichert werden:

```
int x = Jeda.randomInt(20);
```

Aufgabe 6.3

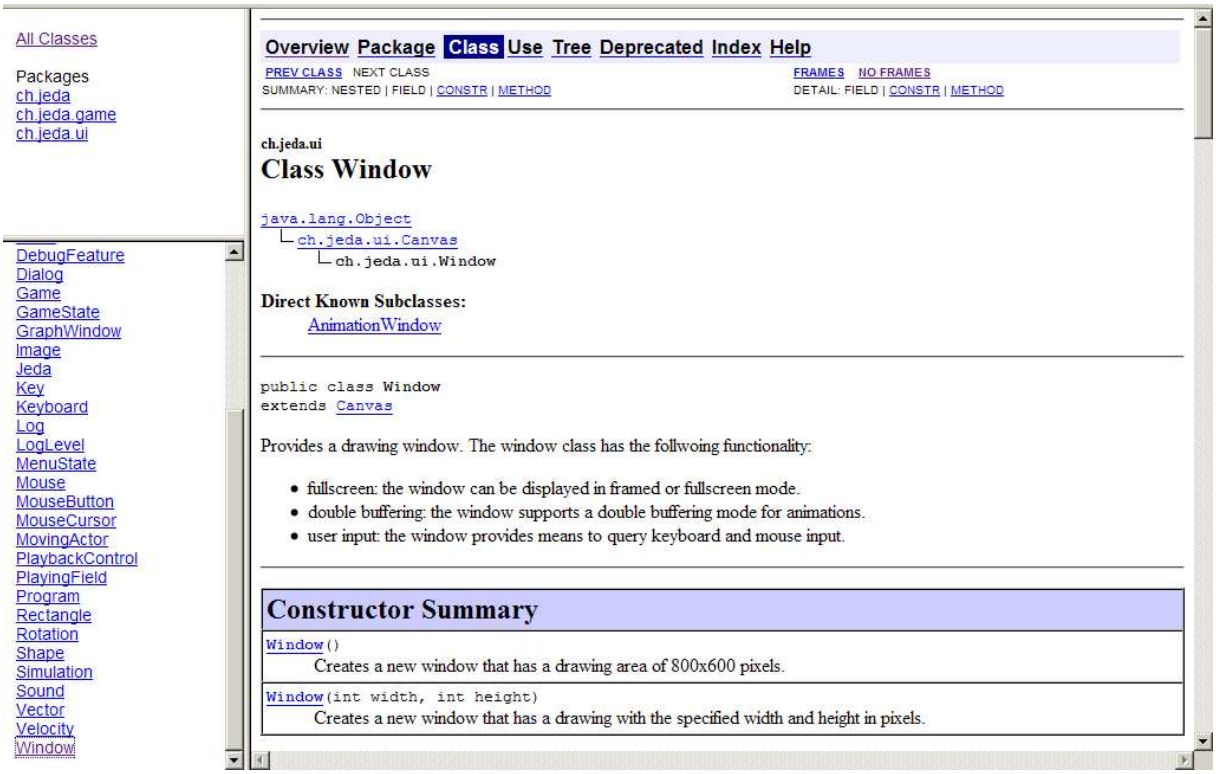
Überlegen Sie sich, wie der Methodenkopf der folgenden Methoden aus der Jeda-Bibliothek aussieht:

- `window.fillRectangle(10, 10, 100, 50);`
- `window.flip();`
- `window.setDoubleBuffered(true);`
- `Dialog.readInt("Gib eine Zahl ein. ");`

6.2.4 JavaDoc

Es ist nicht sinnvoll und meist auch nicht möglich, alles selber zu programmieren. Es gibt eine Vielzahl von zur Verfügung stehenden Bibliotheken, die verwendet werden können. Eine solche Bibliothek ist Jeda. Damit man die Bibliothek verwenden kann, ohne eine Anleitung oder den Quellcode lesen zu müssen, gibt es eine standardisierte Dokumentation für Java-Code (genannt *JavaDoc*).

Auf der Wiki-Seite finden Sie die Java Dokumentation der Jeda-Bibliothek (Unter *Ressourcen*: ***Jeda API-Referenz***). Aber auch die Standardinstallation von Java bietet bereits eine riesige Bibliothek, die in eigenen Programmen verwendet werden kann. Diese Dokumentation ist ebenfalls auf dem Wiki unter *Ressourcen* verlinkt: ***Java API-Referenz***.



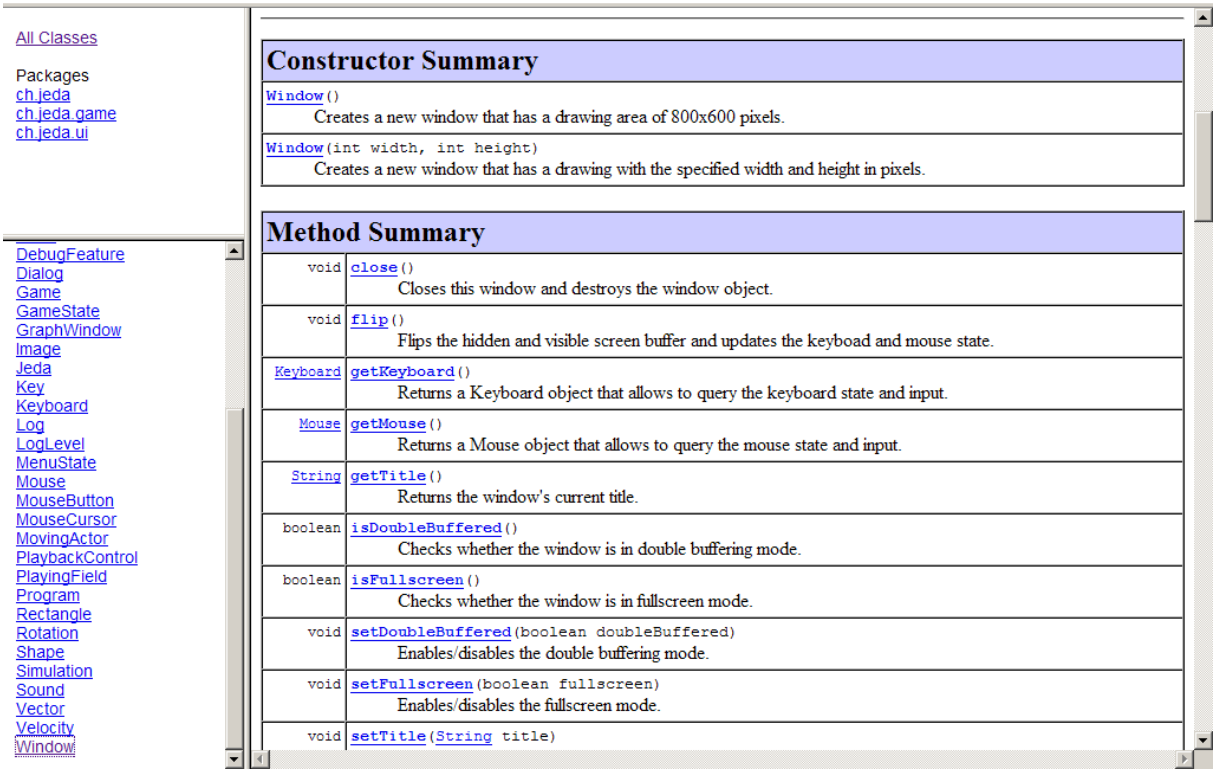
Auf der linken Seite werden sämtliche Klassen der Bibliothek Jeda aufgelistet. Im Hauptfenster rechts wird die Dokumentation der links angeklickten Klasse angezeigt (hier die Klasse ***Window***).

Zuerst folgt eine allgemeine Beschreibung der Klasse. Anschliessend unter der Überschrift ***Constructor Summary***, wie man ein neues Fenster erstellen kann. Die Klasse stellt zwei Möglichkeiten bereit:

- Man erstellt ein neues Fenster durch einen Aufruf wie `window = new Window();`. Die Beschreibungszeile erklärt uns, dass so ein Fenster der Grösse 800x600 Pixel entsteht.
- Alternativ kann man die gewünschte Fenstergrösse auch angeben, indem man `window = new Window(1000, 500);` aufruft.

Den Aufruf mit dem `new`-Operator nennt kann *Konstruktoraufruf*. So wird ein neues Objekt (hier ein Fenster) erzeugt. Deshalb sind die beiden Varianten auch unter dieser Überschrift aufgeführt.

Wenn man etwas weiter nach unten scrollt, findet man unter der Überschrift ***Method Summary*** sämtliche Befehle aufgeführt, die ein Fenster verarbeiten kann:



Eine kurze Beschreibung wird unterhalb eines jeden Befehls angezeigt. Auf der JavaDoc-Seite sieht man nur den Methodenkopf (Signatur) der Methode.

- In einer abgetrennten Spalte wird der Rückgabetyp angegeben (also z.B. `void` also kein Resultat bei der `close()`-Methode oder `String` bei der `getTitle()`-Methode).
- Die Parameter, die man dem Befehl mitgeben muss, werden hinten in der runden Klammer aufgeführt (z.B. `String title` bei der `setTitle()`-Methode). Hier wird – wie im Kapitel 6.2.2 erklärt – der Datentyp ebenfalls aufgeführt. Diese Information ist wichtig, damit man die Methode korrekt aufrufen kann.

Klickt man auf einen Befehl, so erhält man meist noch eine genauere Beschreibung. Die Parameter und auch der Rückgabewert werden dort genauer erklärt.

Aufgabe 6.4

Der User soll vom Programm (Jeda Program Class) nach einer ganzen Zahl gefragt werden und als Antwort erhalten, ob die Zahl gerade ist oder nicht. Vor der Ausgabe soll das Programm allerdings 3 Sekunden warten. Finden Sie den passenden Befehl im JavaDoc der Klasse `Program`.

6.2.5 Verkettung

Die Jeda-Bibliothek ermöglicht auch die Abfrage von Tastatur- und Mauseingaben. Informationen zu gedrückten Tasten liefert die Klasse `Keyboard`. Ein Objekt dieser Klasse (im folgenden Beispiel die Variable `kb`) gibt die gewünschte Auskunft:

```
Keyboard kb = window.getKeyboard();
boolean pressed = kb.isKeyPressed(Key.A);
if (pressed) {
    // Hier steht die Aktion, die ausgeführt werden soll, wenn die
    // Taste A gedrückt wird
}
```

Die ersten beiden Zeilen dieses Beispiels (siehe hervorgehobene Stellen) kann man kürzer schreiben, in dem man die Verkettungsschreibweise nützt. Das Keyboard, das man beim Aufruf von `window.getKeyboard()` als Resultat erhält, muss nicht zwingend in einer Variablen gespeichert werden, bevor man nach einer gedrückten Taste fragen kann. Man dann den Befehl ans Keyboard nämlich direkt mit der Punktnotation hinten anfügen:

```
boolean pressed = window.getKeyboard().isKeyPressed(Key.A);
if (pressed) {
    // Hier steht die Aktion, die ausgeführt werden soll, wenn die
    // Taste A gedrückt wird
}
```

Auch dieser Abschnitt lässt sich noch kürzer schreiben:

```
if (window.getKeyboard().isKeyPressed(Key.A)) {
    // Hier steht die Aktion, die ausgeführt werden soll, wenn die
    // Taste A gedrückt wird
}
```

Aufgabe 6.5

Erweitern Sie das Programm aus der Aufgabe 6.2 (Simulation mit dem an den Wänden abprallenden Ball) so, dass während der Simulation die Farbe des Balles per Tastendruck wie folgt geändert werden kann:

- Zu Beginn ist der Ball schwarz.
- Nach einem kurzen Tastendruck auf den Buchstaben B wird der Ball blau.
- Während dem die Taste G gedrückt gehalten wird, wird der Ball grün gezeichnet, anschliessend aber gleich wieder so, wie vor dem Tastendruck.

Lesen Sie dazu die JavaDoc-Seite der Klasse `Keyboard`. Der Link ist im Wiki unter *Ressourcen* aufgeführt. Diese JavaDoc-Seite sollten Sie normalerweise beim Programmieren geöffnet haben, um jederzeit benötigte Informationen nachlesen zu können.