

Das Raumplanerprojekt

Objekte und Klassen mit BlueJ

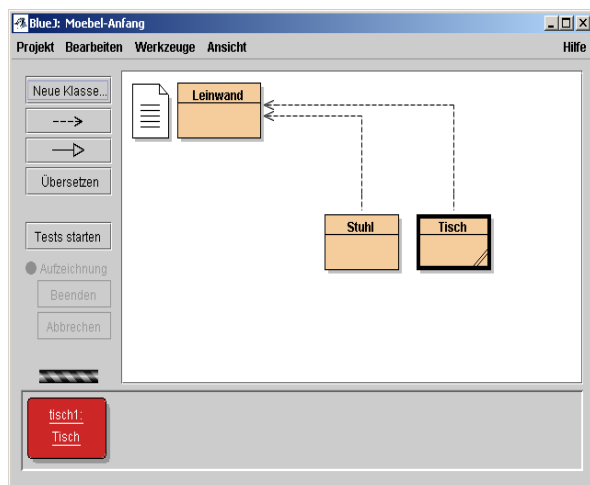
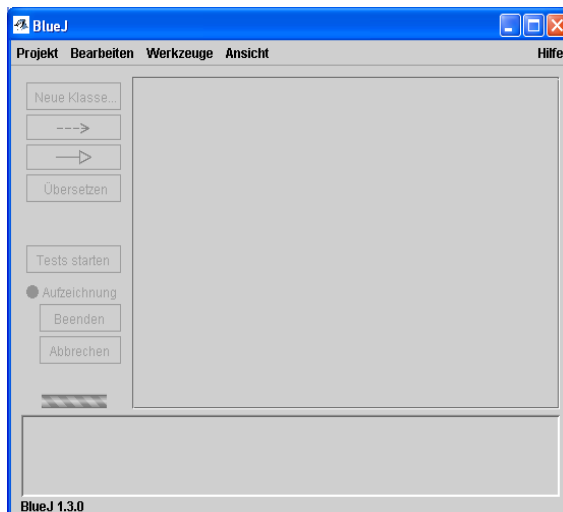
Arbeiten mit der Oberfläche von BlueJ

Die Arbeit mit BlueJ erlaubt einen einfachen Umgang mit Klassen und Objekten in der Programmiersprache Java.

Beim ersten Start sieht die Oberfläche von BlueJ sehr aufgeräumt aus.

Für die ersten Schritte benötigen Sie die Menüleiste, in der man an erster Stelle den Menüpunkt **Projekt** findet.

Jedes JAVA - „Programm“ ist also ein Projekt und benötigt für seine zugehörigen Daten einen eigenen Ordner. Wenn wir eigene Projekte entwickeln, stellt das aber kein Problem dar, da BlueJ jeweils diesen Projektordner selbst erzeugt.



In dem Projekt-Menü finden Sie den Unterpunkt **Projekt öffnen...** über den Sie einen Dialog zum Öffnen von BlueJ-Projekten erhalten. Laden Sie aus dem Ordner des Kurses das Projekt **Möbe1-Anfang** und speichern Sie es unmittelbar danach in Ihren eigenen Ordner. Nach dem Laden dieses Projektes ergibt sich das links dargestellte Bild, wobei sich nach dem Speichern in den eigenen Ordner die Darstellung der orange Rechtecke durch eine hinzugefügte Schraffur geändert hat.

Mit der Schraffur kennzeichnet BlueJ, dass hier noch zu übersetzen ist. Klicken Sie den Button zum Übersetzen (compile) an der linken Seite an und das Bild wird bald den dargestellten Zustand einnehmen.

Im Hauptfenster von BlueJ ist ein Diagramm zu sehen, das durch beschriftete Rechtecke die Klassen des Projektes Möbel und durch Pfeile ihren Zusammenhang darstellt. Der Begriff Klassen ist ein Fachausdruck der Objektorientierung (OO). Da er eines der wichtigsten Konzepte beschreibt, werden wir ihn in Zukunft ständig benutzen. Jetzt zunächst einmal verwenden wir ihn ohne Erläuterung. Was er beschreibt, sollte sich im Verlauf der Arbeit ergeben.

Konkret enthält dies es Projekt die Klassen

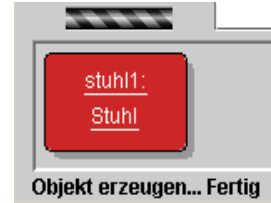
- Stuhl,
- Tisch und
- Leinwand.

Die Pfeile machen deutlich, dass die zwei Möbel-Klassen Stuhl und Tisch die Klasse

Leinwand benutzen. Das Wort „benutzen“ stellt in diesem Zusammenhang einen Fachausdruck der OO dar und zwar für einen Beziehungstyp. Es gibt noch weitere Beziehungstypen bei der OO und wir werden uns mit ihnen und ihrer Unterscheidung beschäftigen.



Zunächst einmal wollen wir uns mit dem Umgang mit BlueJ am Beispiel dieses Projektes beschäftigen. Klicken Sie dazu jetzt mit der rechten Maustaste auf die Klasse `Stuhl` und wählen Sie aus dem Kontextmenü den ersten Menüpunkt `new Stuhl()` aus. Im anschließenden



Dialog (siehe links), der Sie nach einem Namen für die Instanz (das Exemplar bzw. das Objekt – sind jeweils Fachausdrücke der OO mit der selben Bedeutung) von `Stuhl` fragt, können Sie einfach auf OK klicken und damit die Vorgabe `stuhl1` akzeptieren. Es sollte sich jetzt unten im Programmfenster von BlueJ das rechts dargestellte Bild ergeben, mit dem BlueJ kennzeichnet, dass es nun ein Objekt `Stuhl` mit dem Namen `stuhl1` gibt, das ein Exemplar der Klasse `Stuhl` ist. Die Leiste unten im Programmfenster heißt Instanzleiste. Die Schreibweisen sind keine Rechtschreibfehler.

Namenskonventionen in JAVA

In Java gelten folgende Konventionen für die Namen von Klassen und Objekten: Klassen schreibt man mit großem Buchstaben am Anfang des Namens, Exemplarnamen¹ fangen mit Kleinbuchstaben an.

Unbefriedigend ist, dass dies abgerundete Rechteck keineswegs wie ein Stuhl aussieht, das soll es auch gar nicht. BlueJ kennt nun zwar das Objekt `stuhl1`, hat es aber nicht dargestellt. Nicht nur das Programm unterscheidet das, auch wir müssen zwischen einem Objekt, seinem Symbol und seiner Darstellung auf dem Bildschirm unterscheiden. Soll der Stuhl dargestellt werden, dann brauchen wir dafür eine Zeichenfläche. Dass diese schon vorgesehen ist, können wir am Klassendiagramm erkennen. Sie wird aber erst dann dargestellt, wenn eines der darstellbaren Objekte dazu aufgefordert wird.

Methoden aufrufen

Eine solche Aufforderung an ein Objekt zu geben, nennt man „dem Objekt eine Botschaft schicken“ (message) und das programmtechnische Mittel ist der „Aufruf von Methoden“ (Fachausdruck der OO). Um den zugehörigen Stuhl auf den Bildschirm zu bekommen, klicken Sie mit der rechten Maustaste auf das Objekt in der Objektleiste, worauf sich sein Kontextmenü öffnet.



Hier finden Sie alle Methoden, welche die Klasse `Stuhl` zur Verfügung stellt, also alle messages, die sie versteht. Wenn Sie die Methode `zeige()` aufrufen, dann erscheint eine Zeichenfläche und auf ihr das Bild eines Stuhls auf dem Bildschirm und zwar in einem speziellen Fenster, welches zur Klasse Leinwand gehört. Dabei wird der Stuhl in Form eines Umrisses dargestellt, so wie es für

Innenarchitektenzeichnungen vorgeschrieben ist.

¹ Synonyme für „Exemplar“: Instanz oder Objekt

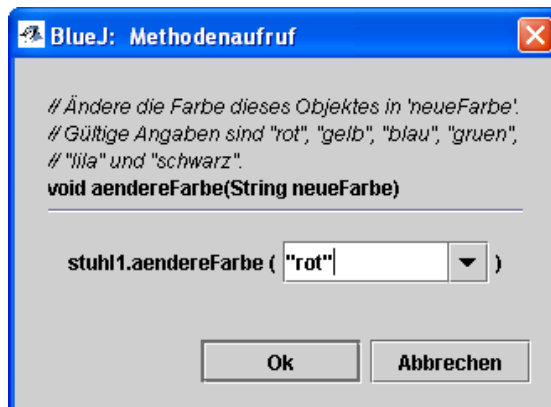
Kontextmenü zum Aufruf von Methoden

Aus dem Kontextmenü von `stuhl1` heraus können Sie weitere Methoden aufrufen und damit z.B. den Stuhl horizontal bzw. vertikal verschieben. Manche dieser Methoden, z.B. `bewegeHorizontal()` erwarten eine Eingabe.

Hier geben Sie eine Zahl ein, z.B. 80, worauf der `stuhl1` sich um 80 Pixel horizontal verschiebt.

Parameter

Etwas aufpassen müssen Sie, wenn Sie die Methode `aendereFarbe` aufrufen, hier erwartet der `stuhl1` keine Zahl, sondern einen Text, der die Farbe angibt. Dieser Text muss in Anführungsstriche gesetzt werden und es stehen auch nur die im Dialogfenster angegebenen Farben zur Verfügung.



Kommentare

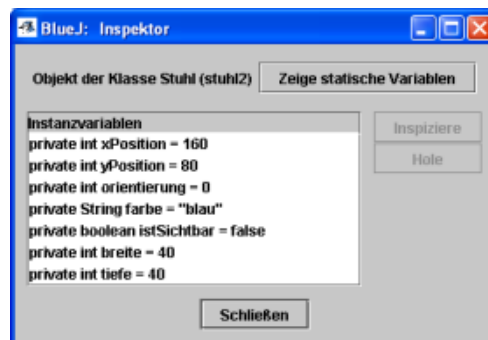
Wie Sie sehen, erleichtert Ihnen BlueJ – und hoffentlich auch derjenige, der den Programmtext geschrieben hat – dies durch einen im Fenster dargestellten Kommentar. Versuchen Sie einmal eine Zahl einzugeben. Sie bekommen dann eine Fehlermeldung, die den falschen Typ bemängelt.

Datentypen in Java

Ein Datentyp beschreibt die *Art* der Information, die Java z.B. als Parameter (ein der Methode übergebener Wert) einer Methode erwartet. Viele der Methoden der Klasse `Stuhl` erwarten Zahlen als Parameter, der zugehörige Datentyp heißt in Java `int`, was eine Abkürzung von `integer`, dem englischen Begriff für ganze Zahl ist. Die Methode `aendereFarbe()` erwartet einen Text, eine Zeichenkette. Der zugehörige Datentyp heißt in Java `String`. Strings oder Zeichenketten müssen Sie generell in Anführungsstriche setzen.

Mehrere Instanzen

Bisher haben Sie nur mit einem einzigen Objekt gearbeitet, nur einer Instanz der Klasse `Stuhl`. Sie können beliebig viele Instanzen der gleichen Klasse erzeugen, aber auch Instanzen verschiedener Klassen gleichzeitig. Bei verschiedenen Instanzen der gleichen Klasse sollten Sie aber unbedingt darauf achten, dass sie unterschiedliche Positionen besitzen, sonst können Sie sie im Leinwandfenster nicht unterscheiden. Sie sind dann zwar vorhanden – wie man in der Objektleiste bei BlueJ erkennen kann – aber nicht erkennbar.



Zustand von Objekten

Bei der Arbeit mit mehreren Objekten, vor allem bei der exakten Positionierung, kann es

immer wieder vorkommen, dass Angaben über einzelne Objekte benötigt, z.B. ihre Position oder ihre Größe. Die Gesamtheit der Werte wie Position, Farbe, Breite, Tiefe, die ein Objekt zu einem Zeitpunkt besitzt, bezeichnet man als seinen Zustand. Der Zustand eines Objektes lässt sich mit BlueJ relativ einfach ermitteln. Im Kontextmenü jedes Objektes steht Ihnen der Menüpunkt Inspizieren zur Verfügung.

Der Zustand eines Objektes der Klasse Stuhl lässt sich durch die angegebenen sieben Datenfelder beschreiben, die hier für die Instanz stuhl2 zu sehen sind. Jede Instanz der Klasse Stuhl verfügt über die gleichen Datenfelder, aber die Werte unterscheiden sich normalerweise von Instanz zu Instanz. Andererseits können sich aber zwei Instanzen der Klasse Stuhl auch nur in diesen Datenfeldern unterscheiden.

Attribute

Eigenschaften von Objekten werden auch als Attribute bezeichnet. Jeder Eigenschaft eines Objektes entspricht ein Datenfeld in der zugehörigen Klasse. Die in ihnen gespeicherten Werte heißen Attributwerte. Sie erfordern unterschiedliche Datentypen. Die Attribute der Klasse Stuhl erfordern die Datentypen `int`, `String`, und `boolean`. Der Datentyp `boolean` wird für Datenfelder verwendet, die nur die Werte wahr (`true`) oder falsch (`false`) speichern müssen. Für das Datenfeld `istSichtbar` wurde der Datentyp `boolean` verwendet, da nur gespeichert werden muss, ob der Stuhl in der Zeichnung sichtbar ist (`true`) oder nicht (`false`).

Aufgabe:

- Erstelle zwei Tische mit jeweils vier Stühlen darum.
- Beobachte dabei, was bei der Arbeit an dieser Aufgabe auf unbefriedigende Funktionalität hindeutet.
- Formuliere Anforderungen an die Funktionalität.
- Versuche die Klassen um diese Funktionalität zu erweitern.

Funktionalität verbessern

Das Verbessern der Funktionalität einer Anwendung ist eine keineswegs triviale Aufgabe. Dabei gilt: Mehr Funktionalität hat nicht notwendig auch ein erfolgreicherer Arbeiten zur Folge. Dies hat auch bei den Entwicklern von Software zu einem Umdenken geführt.

Entwickelt man für einen Kunden Software, dann sollte sie zwar prinzipiell erweiterbar sein, sie sollte aber noch nicht alle möglichen Aufgaben berücksichtigen, zu denen der Kunde noch keine Wünsche geäußert hat, von denen der Entwickler aber meint, der Kunde könnte sie später noch wünschen. Funktionalität wird nur dann vorgesehen, wenn der Kunde sie wünscht und Vorüberlegungen dazu vermeide man. Das geschieht dann auch auf die Gefahr hin, dass man später zu einem redesign gezwungen ist.

Dennoch lässt sich für die vorliegende Anwendung feststellen: Die Klassen Stuhl und Tisch haben für den Benutzer unzureichende Funktionalität.

Erzeugt man nämlich mehrere Exemplare dieser Klasse und stellt sie dar, erhält man das unbefriedigende Ergebnis, dass alle – in ihren Eigenschaften gleichen – Exemplare sich überdecken, wie eines aussehen, also optisch nicht unterscheidbar sind, obwohl sie selbstverständlich nicht dieselben Objekte sind.

Allgemein gilt, dass verschiedene Objekte durchaus in allen ihren Attributwerten übereinstimmen dürfen, ohne dass sie damit dasselbe Objekt sind. Gerade bei Möbelstücken kann das an sich sinnvoll sein, wenn es z.B. nur einen Typ gibt. Allerdings stehen sie aber sicher nicht alle am selben Platz und sollten in diesen Attributwerten verschieden sein. Daher wäre es vorteilhaft, bereits beim Erstellen von neuen Stühlen und Tischen die Position, deren Größe, usw. beeinflussen zu können.

alternative Konstruktoren

So kann es sinnvoll sein, in der Klasse einen weiteren Konstruktor anzubieten, der zwar die selben Standardwerte von Orientierung, Farbe, Länge und Breite vorsieht, aber beim Erzeugen des Exemplars die Übergabe einer unterscheidbaren Position vorsieht.

Die Methode einer Klassendefinition, die man zum Erstellen eines Exemplars verwendet, also im Fall der Klasse Stuhl, um ein Objekt wie stuhl1 zu erzeugen, heißt Konstruktor (Fachausdruck). Der vorgegebene Konstruktor erzeugt ein Objekt mit vollständig vorgegebenen Standardwerten, erstellt man zwei, dann sind sie optisch nicht unterscheidbar, da einer den anderen exakt verdeckt. Daher wäre es schön, wenn man in der Anwendung die Möglichkeit hat, den zweiten Stuhl gleich mit einer anderen Position zu erzeugen.

Überladen

Java bietet die Möglichkeit, Methoden und Konstruktoren zu definieren, die bei gleichem Namen trotzdem auf unterschiedliche Arten aufgerufen werden können. Diese Technik bezeichnet man als Überladen. Wir untersuchen diese Möglichkeit am Beispiel der Konstruktoren.

Konstruktor

Ein Konstruktor einer Klasse ist eine Initialisierungsmethode für ein neu zu erzeugendes Objekt. Der Konstruktor kann Parameter mit der Bedeutung von Initvariablen haben. Initvariable sind Variable für Startwerte (Initwerte) von Attributen.

Wenn sich unsere Klasse ändern soll, müssen wir den zu ihr gehörenden Text verändern. Dazu doppelklicken wir in BlueJ auf das Klassensymbol und es erscheint ein Textfenster mit dem Text der Klassendefinition. Diesen können wir nun bearbeiten.

Nach dem Bearbeiten müssen wir unser Ergebnis übersetzen (compile) und können dann – hoffentlich – mit der neuen Variante arbeiten.

Durch die Möglichkeit des Überladens brauchen wir den bisherigen Konstruktor nicht zu ersetzen, wir fügen der Klasse einfach einen weiteren, sogar mit dem selben Namen hinzu.

Zum vorgegebenen Konstruktor gehört folgender Programmtext:

```
/**
 * Erzeuge einen neuen Stuhl mit einer Standardfarbe und
 * Standardgroesse an einer Standardposition.
 */
public Stuhl()
{
    xPosition = 160;
    yPosition = 80;
    farbe = "blau";
    orientierung = 0;
    istSichtbar = false;
    breite = 40;
    tiefe = 40;
}
```

Die Klasse Stuhl verwendet also für das Speichern der Eigenschaften (Attribute) die Felder (Variablen) xPosition, yPosition, farbe, orientierung, istSichtbar, breite und tiefe. Auch hier, bei den Attributen finden wir eine allgemein verwendete Schreibweise: Die Attributnamen beginnen alle mit einem kleinen Buchstaben, setzt sich der Name aus mehreren Worten zusammen, werden sie zusammenhängend geschrieben – damit das System nicht meint, es seien mehrere gemeint – und der Anfangsbuchstabe der nachfolgenden Teilworte wird jeweils mitten im Wort groß geschrieben.

Die Parameter, die wir einbringen wollen, müssen in der Klammer nach dem Konstruktornamen auftauchen. Wollen wir also die Stühle mit verschiedenen Startpositionen erzeugen können, müssen für diese Startwerte (Initwerte) in der Klammer Initvariable angegeben werden. Der Kopf unseres Konstruktors könnte dann so aussehen (Namen kann man frei wählen):

```
public Stuhl(int initX, int initY)
```

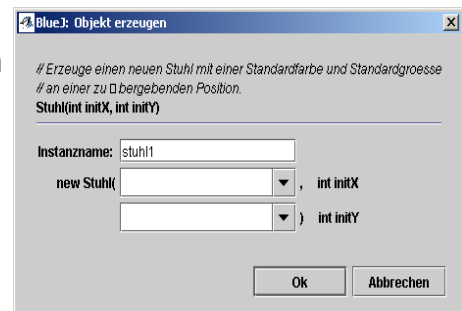
int kennzeichnet wieder den Datentyp. Diese Werte müssen nun beim Erzeugen eines Objektes den Objektvariablen zugewiesen werden:

```
xPosition = initX;
yPosition = initY;
```

Das war schon alles. Sollten wir keine Fehler gemacht haben, können wir die geänderte Klasse nun übersetzen und anschließend benutzen.



Beim Erzeugen des Objektes haben sich nun die Fenster geändert: Wir können mit dem zweiten bereitgestellten Konstruktor die Startwerte für x und y übergeben.






















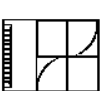


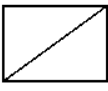

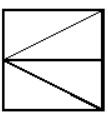







Erweiterung der Funktionalität durch weitere Methoden

Für weitere Methoden lassen sich viele Beispiele finden. Man könnte z.B. für den Raum ein Raster vorgeben, an dem sich alle Moebelstücke orientieren müssen. Ein Verschieben wäre dann nur um Vielfache dieses Rasters möglich. Interessant wäre an dieser Stelle auch, das Überladen bei Methoden wie z.B. `bewegeHorizontal` einzubauen, indem diese Methode ohne Parameter aufgerufen werden kann und dann um eine vorgegebene Weite weitersetzt.

Wichtig ist an dieser Stelle, dass Sie lernen, mit BlueJ und den Grundelementen der Sprache JAVA umzugehen. Neue Sprachelemente werden dann besprochen, wenn man auf sie trifft. Eine weitere Aufgabe mit ähnlicher Zielsetzung ist:

Aufgabe 2

Erstelle weitere Möbelklassen mit einem Erscheinungsbild, das den Normzeichen entspricht: Schrank, Bett, Regal, Sessel, Sofa, usw.

Wohnen			Bad		
	Tisch	80x120		Badewanne	180x80
	Runder Tisch	100 Ø		Dusche	80x80
	Schreibtisch	60x180		Spülklosett	40x60
	Schreibtischstuhl	40x40		(mit Spülkasten)	40x70
	Sofa	80x150		Handwaschbecken	60x50
	Stuhl, Sessel	45x45 60x60		Waschmaschine	60x60
	Hocker	40 Ø		Heizkörper	100x10
	Schrankelement (Hochschrank)	60x110	Kochen		
	Sidebord, Anrichte	50x100		Geschirrspülmaschine	60x60
	Klavier	60x140		Elektroherd (mit Backofen)	60x60
	Regal	200x150		Gasherd (mit Backofen)	60x60
Schlafen				Kühlschrank	60x60
	Franz. Bett	160x220		Gefrierschrank	60x60
	Doppelbett	210x210		Arbeitsfläche mit 2 Oberschränken	30-60x100
	Nachtisch	40x40		Spülbecken mit Abtropffläche	100x60
	Fernseher	45x50		Warmwasserbereiter (Elektro/Gas)	50 Ø
	Garderobe	25x140		Blumentische	40x40 40 Ø

Das Zeichnen weiterer Möbelsymbole

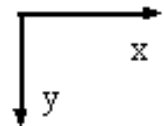
In unserem Raumplaner fehlen noch viele Klassen. Wenn wir sicher auch nicht Vollständigkeit anstreben werden, so geht es uns aber darum, die grundsätzlichen Konzepte des Zeichnens in JAVA zu verstehen. In unseren beiden vorliegenden Klassen sind zwei etwas von einander abweichende Konzepte realisiert, die wir uns ansehen müssen. Beginnen wir mit der Klasse Stuhl. In ihrem Programmtext finden wir eine Methode `gibAktuelleFigur()`, die ganz offensichtlich die Zeichnung definiert:

```
/**
 * Berechnet das zu zeichnende Shape anhand der gegebenen Daten
 */
private Shape gibAktuelleFigur()
{
    GeneralPath stuhl = new GeneralPath();
    stuhl.moveTo(0 , 0);
    stuhl.lineTo(breite, 0);
    stuhl.lineTo(breite+(breite/20+1), tiefe);
    stuhl.lineTo(-(breite/20+1), tiefe);
    stuhl.lineTo(0 , 0);
    //Das ist die Umrandung. Das Stuhl bekommt noch eine Lehne
    stuhl.moveTo(0 , (breite/10+1));
    stuhl.lineTo(breite, (breite/10+1));

    // transformieren
    AffineTransform t = new AffineTransform();
    t.translate(xPosition, yPosition);
    Rectangle2D umriss = stuhl.getBounds2D();
    t.rotate(Math.toRadians(orientierung) ,
            umriss.getX()+umriss.getWidth()/2,
            umriss.getY()+umriss.getHeight()/2);
    return t.createTransformedShape(stuhl);
}
```

Die Ähnlichkeit des ersten Methodenabschnittes zu Beschreibungen in LOGO – ähnlichen Systemen mit turtle - Grafik ist so deutlich, dass das Verständnis nicht schwer fallen sollte.

- `moveTo(xPos, yPos)` ist eine Methode, die den Zeichenstift auf die durch $(xPos, yPos)$ definierte Bildschirmposition *setzt*, ohne dass dabei gezeichnet wird.
- Das Grafiksystem verwaltet also offensichtlich die aktuelle Position des Zeichenstiftes.
- `lineTo(xPos, yPos)` ist eine Methode, die von dieser aktuellen Position des Zeichenstiftes des Zeichenstift eine Linie zur durch $(xPos, yPos)$ definierten Bildschirmposition *zeichnet*.
- Positionen werden in Bildschirmkoordinaten angegeben. Das System beginnt in der linken oberen Ecke unserer Zeichenfläche im Punkt $(0,0)$, die x – Richtung geht nach rechts und –ungewohnt– die y – Richtung nach unten.



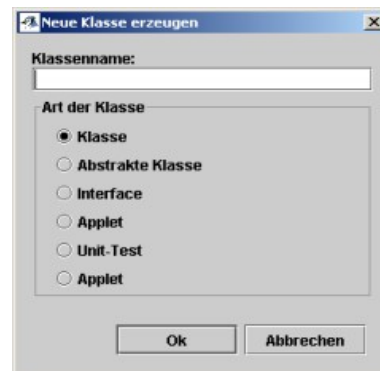
Mit dem im zweiten Abschnitt des in der Methode folgenden Programmtextes wollen wir uns zunächst noch nicht beschäftigen. Diese betrachten wir zunächst genau so als eine „black box“, wie die Klasse Leinwand. Wir akzeptieren, dass sie das tun, was wir wollen und heben uns eine Klärung, wie das geschieht, für später auf.

Das Zeichnen eines Sofas, Sessels, Schrankelements oder Bettes sollte nach dieser Kenntnis nicht schwer fallen.

Zwei Fragen bleiben aber:

- Wie erstelle ich die neue Klasse?
- Wie wird der Tisch gezeichnet?

Die neue Klasse erstellt man am einfachsten in zwei Schritten. Zunächst einmal fordert man BlueJ auf, eine neue Klasse zu erstellen. Im Dialogfenster gibt man den Namen, z.B. Schrank an, behält die Einstellungen bei und drückt dann OK. Im Klassenfenster taucht nun ein weiteres Rechteck als Symbol für die neue Klasse auf, das als Schrank beschriftet ist. Nun öffnet man die Programmtexte der Klassen Stuhl und Schrank, markiert bei Stuhl den ganzen Text, kopiert ihn, markiert in Schrank den gesamten Text und überschreibt ihn durch Einfügen mit dem von Stuhl.



Bevor man etwas anderes macht, korrigiert man in dieser Klasse nun die Namen in der Klassendefinition und in den Konstruktoren auf den neuen Namen Schrank. (Sinnvollerweise korrigiert man auch die Namen in den Kommentaren.) Erst jetzt geht man daran, den Text der Methode `gibAktuelleFigur()` zu korrigieren.

Die zweite Frage führt auf ein völlig neues Problem. Mit Hilfe von sehr vielen kleinen Linienabschnitten könnte man natürlich das Problem lösen, eine gebogene Linie zu zeichnen. JAVA bietet uns aber eine andere Möglichkeit, die wir hier nutzen. Den Hinweis gibt uns der Datentyp, den der Rückgabewert der Methode `gibAktuelleFigur()` hat. Da wir aber über Rückgabewerte von Methoden uns noch keine Gedanken gemacht haben, wird das hier notwendig. Unsere einfache Klassendefinition von Stuhl enthält nur an dieser Stelle einen Rückgabewert.

`private` im Kopf der Methode kennzeichnet den Typ der Sichtbarkeit, sie ist in diesem Fall auf die Klasse Stuhl beschränkt. `shape` dagegen gibt den Datentyp des Rückgabewertes an. Die Methode muss dann eine `return` - Anweisung mit einem `shape` - Wert enthalten. Die folgende Methode dagegen:

```
/**
 * Mache dieses Objekt sichtbar. Wenn es bereits sichtbar ist, tue nichts.
 */
public void zeige()
{
    istSichtbar = true;
    zeichne();
}
```

hat keinen Rückgabewert. Außerdem ist sie vom Sichtbarkeitstyp `public` und ist daher nach außen voll sichtbar, also auch im Kontextmenü der Objekte. Dass sie keinen Rückgabewert hat, kennzeichnet man mit `void`. Eine `return` - Anweisung ist nicht sinnvoll. Unsere Methode `gibAktuelleFigur()` liefert ein Objekt zurück, das als `shape` gekennzeichnet ist. Da eine solche Klasse nicht selbstverständlich ist, müssen wir in der JAVA - Klassendokumentation nachsehen, was ein `shape` ist. Dort heißt es:

```
java.awt
Interface Shape
All Known Implementing Classes:
...
```

interface

Shape ist überraschenderweise selbst keine Klasse, sondern ein interface, zu deutsch Schnittstelle. Von einem Shape selbst kann kein Objekt erzeugt werden. Das interface beschreibt nur, welche Eigenschaften alle Klassen haben müssen, die dieses interface implementieren. Implementieren heißt, dass sie sich nach außen hin so verhalten müssen, wie man es von einem Shape erwartet. Was man erwartet, ist in der Definition des interfaces abgelegt.

abstrakte Klasse

Wir finden hier aber leider nicht die Klasse angegeben, welche die konkrete Implementation von Shape im Fall des Tisches ist, das ist nämlich die Klasse `Ellipse2D`. Bei `Ellipse2D` tritt auch das Problem auf, dass wir keine Objekte davon erzeugen können. `Ellipse2D` ist eine abstrakte Klasse. Erst die von ihr abgeleiteten Klassen `Ellipse2D.Double` und `Ellipse2D.Float` sind „normale“ Klassen, von denen wir Objekte erzeugen können. Mit den genaueren Hintergründen werden wir uns noch beschäftigen. Hier sollte zunächst die Information reichen.

Objekte vom Typ Shape?

Erstaunlicherweise gibt also die Methode `gibAktuelleFigur()` Objekte vom Typ Shape zurück, obwohl es gar keine Objekte gibt, die „reine“ Shapes sind, also nur Shape und nichts anderes. Wir könnten prinzipiell ebenso auch mit `Ellipse2D` verfahren. Das System braucht nicht zu wissen, zu welcher konkreten Klasse das Shape gehört. An dieser Stelle muss die Information reichen, dass es ein Shape ist.


Dies scheint zunächst ein Informationsverlust zu sein. Tatsächlich ermöglicht es aber der Methode `gibAktuelleFigur()` immer ein Shape zu liefern, egal, was der konkrete Typ ist. Greift eine andere Methode auf diesen Wert zu, muss sie ihn allerdings auch wie ein Shape behandeln und nicht wie ein `Rectangle`, `Line` oder anderes. Die in der letzten Zeile aufgerufene Methode `createTransformedShape(<name des speziellen Shape>)` macht genau das und liefert auch wieder ein Shape zurück. Sie arbeitet also z.B. für „tisch1“, der ein `Ellipse2D.Double` – Objekt ist, aber auch für einen „stuhl1“, der ein `GeneralPath`¹ – Objekt ist.

Umriss

Die Methode verwendet für die Bestimmung des Drehzentrums für die Drehung (Rotation) die Eigenschaft von `shapes`, dass sie den von ihnen ausgefüllten rechteckigen Bereich des Bildschirms kennen. Da wir alle Möbelobjekte immer als `shapes` und immer zunächst waagrecht ausgerichtet definieren, lässt sich zu ihnen sehr einfach dieses Rechteck angeben. Dieses Rechteck selbst, das in unserem Projekt als `Umriss` bezeichnet wurde, ist übrigens auch ein `shape`, nämlich ein `Rectangle2D`². Auch bei zusammengesetzten Figuren kann man so einfach das Drehzentrum definieren.

1 Anmerkung: Mit `Polygon` lässt sich genau so ein geschlossener Linienzug zeichnen, wie mit den Methoden von `GeneralPath`, die wir bei Stuhl verwendet haben. Es gibt weitere Überschneidungen.
2 Das hat zur Folge, dass wir das entsprechende Paket im Kopf der Klassendefinition über `import java.awt.geom.Rectangle2D;` importieren müssen.

Ein Beispiel für einen weiteren Möbeltyp

Einige der Möbelzeichnungen lassen sich nur aufwändig aus mehreren Teilfiguren zusammensetzen. Für solche Figuren ist die Klasse `GeneralPath` sehr hilfreich, da sie nicht nur elementare Zeichenmethoden wie `moveTo(...)` und `lineTo(...)` bereitstellt, sondern über die `append(...)` – Methode auch andere shapes aufnehmen kann. Als Beispiel verwenden wir die Badewanne,  die man aus mehreren Teilfiguren zusammensetzen kann, nämlich

- einem vollständigen Rechteck,
- einem unvollständigen Rechteck, das wir aus drei Linien mit einem Halbkreis (Klasse `Arc`) daran zusammensetzen werden
- und einem kleinen Kreis für den Ablauf.

Für die zu verwendenden Maße nehmen wir relative Werte, also Angaben, die abhängig von den Werten `breite` und `tiefe` sind. Vorgegebene Maße aus der Normzeichnung sind `180 x 80`, das definiert uns die Werte für das volle Rechteck. Für das innere, unvollständige Rechteck lassen wir auf jeder Seite gleich viel, z.B. ca. 10% der Tiefe weg, so dass dessen Position in der Figur nicht durch die linke obere Ecke `(0;0)`, sondern durch `(0.9*tiefe;0.9*tiefe)` beschrieben wird, seine Tiefe durch `0.8*tiefe` (oben und unten weniger!). Die Breite ist komplizierter zu bestimmen, da auf der rechten Seite auch der Radius des Halbkreises abgezogen werden muss.

Man muss bei einer solchen Figur sicher etwas probieren, bis man passende Werte findet.

Mit:

```
GeneralPath badewanne = new GeneralPath();
Rectangle2D umriss = new Rectangle2D.Double(0, 0, breite, tiefe);
badewanne.append(umriss, false);
Line2D obererRand = new Line2D.Double(0.1*tiefe, 0.1*tiefe, breite - 0.5*tiefe,
    0.1*tiefe);
Line2D linkerRand = new Line2D.Double(0.1*tiefe, 0.1*tiefe, 0.1*tiefe,
    0.9*tiefe);
Line2D untererRand = new Line2D.Double(0.1*tiefe, 0.9*tiefe, breite - 0.5*tiefe,
    0.9*tiefe);
Arc2D bogen = new Arc2D.Double(breite - 0.9*tiefe, 0.1*tiefe, 0.8*tiefe,
    0.8*tiefe, 270, 180, Arc2D.OPEN);
badewanne.append(obererRand, false);
badewanne.append(linkerRand, false);
badewanne.append(untererRand, false);
badewanne.append(bogen, false);
Ellipse2D ablauf = new Ellipse2D.Double(0.1*breite, 0.5*tiefe-2, 4, 4);
badewanne.append(ablauf, false);
```

erhält man eine schöne Badewanne!

Beim `append` muss als zweiter Parameter jeweils `false` angegeben werden, damit zwischen den einzelnen Teilen der Figur keine Verbindungslinien gezeichnet werden.

