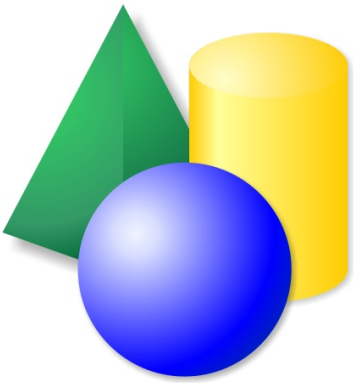
Eine Kurzeinführung in



Kids' Programming Language

Vorwort

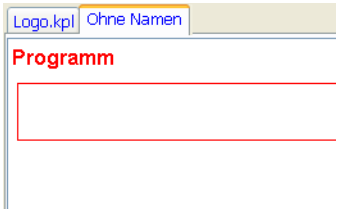
Die vorliegende Kurzeinführung in die Programmiersprache Kids' Programming Language (deutsch) gibt eine Übersicht über die Syntax der Sprache. Sie richtet sich damit an erfahrene Programmierende und nicht an Programmieranfänger(innen). Trotzdem wurde die Beschreibung der Programmiersprache fast ausschließlich anhand von repräsentativen Beispielen durchgeführt und auf eine syntaktische Definition verzichtet. Bitte beachten Sie, dass für das tiefere Verständnis einiger Kapitel Kenntnisse über die Grundlagen der objektorientierten Programmierung empfehlenswert sind. Unterkapitel mit erhöhten Anforderungen, die beim ersten Lesen übergangen werden können, wurden mit einem Stern (*) gekennzeichnet.
Neben der reinen Syntax der Sprache ist für das praktische Programmieren wichtig, welche "Klassen" zur Verfügung stehen. Da dies in dieser Kurzeinführung nur am Rande erwähnt wird, konsultieren Sie dazu bitte die "Klassenreferenz", in der die Methoden aller Klassen mit vollständiger Beschreibung der Parameter aufgeführt sind.
Im ersten Kapitel wird die Programmierumgebung (die Programmieroberfläche) von Kids' Programming Language beschrieben. Dieses Kapitel greift inhaltlich an vielen Stellen späteren Kapiteln voraus, gibt aber anderseits einen wichtigen Überblick über die Funktionalität der Programmierumgebung. Es wird daher empfohlen dieses Kapitel zunächst nur oberflächlich zu lesen und später durchzuarbeiten.

Inhalt

1. Kids' Programming Language verwenden

Die Programmierumgebung

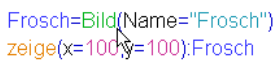
Das Editorfenster



In der Mitte der Programmierumgebung erscheint beim Starten von Kids' Programming Language ("KidsPL") bereits ein (leeres) Programm mit dem Titel "Ohne Namen". Es können gleichzeitig mehrere Programme bearbeitet werden und durch Drücken auf die Registrierkartentitel kann zwischen den Programmen gewechselt werden.

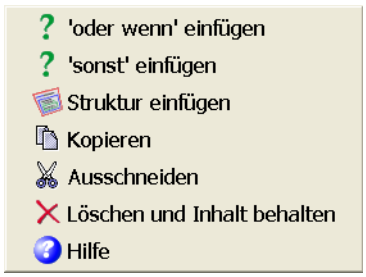
In das Programm lassen sich Anweisungen über Tastatur eingeben oder über einen Assistenten einfügen (siehe unten). Die Sprachelemente werden zur besseren Übersichtlichkeit farblich hervorgehoben. Die Farbzuordnungen entnehmen Sie bitte der folgenden Tabelle. Beachten Sie, dass die aufgelisteten Sprachelemente erst später beschrieben werden.

dunkelblau	Symbole =(),:~+-*/><
hellblau	Objekte
grün	Klassennamen für neue Objekte
orange	Methodenaufrufe
türkis	Zeichenketten
violett	Zahlen
dunkelviolett	Kommazahlen
dunkelblau, fett	Schlüsselwörter
rot, fett	Klassenname, Methodenname
grün, fett	Bedingungen (wenn...dann)
roter Rahmen	um Klassen und Methoden
grüner Rahmen	um Bedingungen
blauer Rahmen	um Schleifen
grau	Kommentar
rot, unterstrichen	fehlerhafte Stellen



Der Text einer Zeile, die gerade editiert wird, erscheint in schwarzer Schrift. Werden die ↵-Taste oder die Cursortasten "Nach oben" und "Nach unten" benutzt, wird die Zeile formatiert und sofort auf Fehler überprüft. Durch einen Doppelklick auf ein Sprachelement lässt sich die Kontexthilfe im Hilfe-Browser (siehe unten) aufrufen. Man erhält Informationen über **Schlüsselwörter** (fett), Methoden (orange), Klassen (grün) und Objekte (blau).

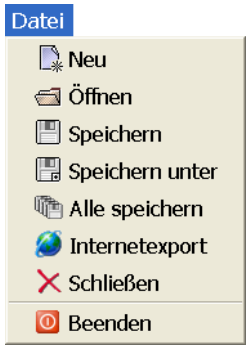
Ein Klick mit der rechten Maustaste öffnet ein Popup-Menü, mit dessen Hilfe man Text und Strukturen einfügen, kopieren und ausschneiden kann. Außerdem lässt sich die Hilfe aufrufen.



Wird auf das Schlüsselwort einer Struktur (wenn, wiederhole, solange, zähle, benutze und Methode) mit rechten Maustaste geklickt, wird ein spezifisches Kontextmenü angezeigt. Es bietet unter anderem die Möglichkeit Strukturen außen herum einzufügen oder zu löschen und den Inhalt der Struktur zu behalten. Bei einer 'wenn...dann' Struktur lassen sich insbesondere 'oder wenn' und 'sonst' einfügen.

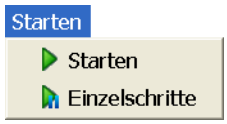
Menüs

Über dem Editorfenster befindet sich eine Menüleiste mit den Einträgen "Datei", "Bearbeiten", "Starten", "Assistent", "Hilfe" und "Hilfe zu".

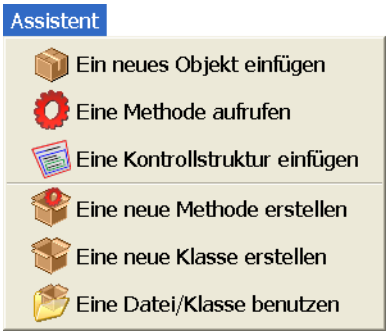


Unter dem Menü "Datei" haben Sie die Möglichkeit Dateien zu öffnen ("Öffnen") und zu speichern ("Speichern"). Drücken Sie zum Erzeugen eines neuen Programms auf "Neu" und zum Schließen des aktuellen Programms auf "Schließen". Um ein bereits gespeichertes Programm unter einem anderen Namen zu speichern wählen sie "Speichern unter". KidsPL Programme lassen sich außerdem im Internet publizieren ("Als Internetseite speichern"). Um die Programmierumgebung von Kids Programming Language zu schließen wählen sie "Beenden".

Sie können die Zwischenablage von Windows benutzen und Textpassagen "Ausschneiden", "Kopieren" und "Einfügen". Außerdem lassen sich Änderungen am Programm "Rückgängig" machen oder "Wiederherstellen".



Unter dem Menüpunkt "Starten" lässt sich das aktuelle Programm "Starten". Des Weiteren lässt sich das Programm Schritt für Schritt (im so genannten Debugger) ausführen.



Programmieren per Mausklick erlaubt der Programmier-"Assistent": Dialoggestützt lassen sich "Kontrollstrukturen einfügen" (wenn...dann, wiederhole, solange...mache und zähle...mache) sowie "Objekte einfügen" und "Methoden aufrufen". Dem Programm kann außerdem eine neue Klasse ("Eine neue Klasse erstellen") oder eine Methode ("Eine neue Methode erstellen") hinzugefügt werden. Zusätzlich können auch Klassen aus anderen Dateien benutzt werden ("Eine Datei/Klasse benutzen"). Diese Menüpunkte finden sich auch in der Aktionsleiste (siehe unten) und werden dort detailliert beschrieben.

Einige der Menüpunkte lassen sich direkt über die Leiste unter dem Menü aufrufen. In der Leiste sind nur die Icons ohne Text abgebildet:

Halten Sie den Mauszeiger eine kurze Zeit über ein Icon um Hilfe zu einem Symbol zu erhalten.

Aktionsleiste

An der linken Seite der Programmierumgebung befindet sich die Symbolleiste des Programmier-Assistenten. Im Folgenden werden die möglichen Aktionen beschrieben. Die genaue Erklärung der Syntax erfolgt an späterer Stelle.

Ein neues Objekt einfügen:
Durch das Drücken dieser Taste wird ein Assistent zum Einfügen eines neuen Objekts aufgerufen. Sie können die Klasse des Objekts auswählen sowie den Namen und die Parameter bestimmen. Außerdem ist es dabei möglich (kontextabhängig) auf bereits eingefügte Objekte zurückzugreifen.

Eine Methode aufrufen:
Fügt einen Methodenaufruf an der aktuellen Stelle im Editor ein. Sie können aus einer Liste ein Objekt sowie den Namen der Methode auswählen und zusätzliche Parameter bestimmen. Auch der Methodenaufruf eines neuen Objektes ohne Namen (anonyme Objekte) ist möglich.

Wenn ... dann:
Wenn die Bedingung ... erfüllt ist, wird der Inhalt der Struktur ausgeführt.

Wiederhole:
Der Inhalt der Struktur wird beliebig oft wiederholt.

Solange ... mache:
Der Inhalt wird wiederholt, solange die Bedingung ... erfüllt ist.

Zähle ... mache:
Der Inhalt wird so oft wie angegeben wiederholt (z.B. zähle bis 10 mache). Der aktuelle Zählerstand kann in einer Zahl gespeichert werden (z.B. zähle i von 2 bis 42 mache).

Eine neue Methode einfügen:
Fügt eine Methode in die aktuelle Klasse ein. Name, Parameter und Rückgabewert können über einen Dialog bestimmt werden.

Eine neue Klasse einfügen:
Fügt eine neue Klasse in das Programm ein. Name und Parameter lassen sich über einen Dialog bestimmen.

Eine Datei/Klasse benutzen:
Bindet Klassen aus anderen Dateien ein. Die Dateien lassen sich über einen Dialog wählen und sind relativ zum Pfad "Eigene Dateien\KidsPL" beziehungsweise "Benutzer/KidsPL". Beginnt der Pfad mit "/" handelt es sich um eine absolute Pfadangabe und bei "./" um eine Pfadangabe relativ zur aktuellen Datei.

Klassen-Browser

Im Klassen-Browser werden die Elemente des Programms (Klassen, Methoden, Objekte) strukturiert als Baum dargestellt. Es werden dabei alle Objekte angezeigt, die an der aktuellen Stelle des Cursors verwendet werden können. Folglich verändert sich der Inhalt des Klassen-Browsers, je nachdem in welcher Zeile sich der Cursor befindet. Klassenweite Objekte erscheinen unter dem Klassennamen, während lokale Objekte unter der zugehörigen Methode eingeblendet werden.

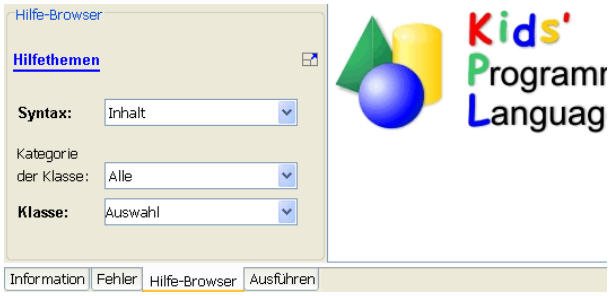
Durch einen Doppelklick auf "Neues Objekt" lässt sich ein neues Objekt an der aktuellen Stelle im Editor einfügen. Das Objekt erscheint mit seinem Namen anschließend (in blauer Farbe) im Klassen-Browser.

Ein Doppelklick auf ein Objekt (blau) bewirkt, dass die Stelle, an der dieses Objekt definiert wird, im Editorfenster angezeigt wird. Mit einem Klick auf das "+" vor einem Objekt lassen sich die Methoden des Objektes (orange) anzeigen. Durch einen Doppelklick auf eine Methode wird diese Methode an der aktuellen Stelle eingefügt. Ein einfacher Klick auf das Objekt oder die Methode öffnet die Kontexthilfe im Hilfe-Browser.

Der Klassen-Browser zeigt neben den Objekten auch die Klassen und Methoden des geschriebenen Programms an. Die aktuelle Klasse und Methode werden in rot angezeigt, andere erscheinen in schwarzer Farbe im Klassen-Browser. Ein Doppelklick auf eine Klasse oder eine Methode bewirkt, dass diese Klasse oder Methode im Editorfenster angezeigt wird. Zusätzliche Informationen wie Parameter oder geerbte Methoden sind grau und lassen sich mit einem Klick auf "+" erweitern.

Hilfe-Browser

Im Hilfe-Browser können Informationen zu der Syntax oder zu Klassen und Methoden angezeigt werden. Dazu muss unter "Syntax" oder "Klasse" nur der gewünschte Eintrag ausgewählt werden. Das Feld "Kategorie" dient dazu, bestimmte Klassen herauszufiltern, damit zum Beispiel unter "Klassen" nur Klassen aufgeführt werden, mit denen sich Grafikobjekte darstellen lassen. Eine Übersicht über die Hilfethemen bekommt man durch einen Klick auf "Hilfethemen". Außerdem lässt sich mit Hilfe des Symbols  der Hilfe-Browser in einem neuen und größeren Fenster öffnen.

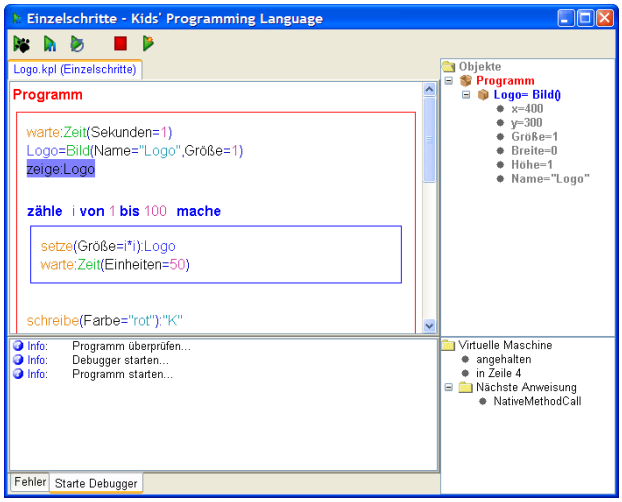


Eine Liste der Hilfethemen erhalten Sie übrigens durch einen Klick auf "Hilfethemen". Außerdem werden unter den Registrierkarten "Fehler" und "Ausführen" aktuelle Meldungen angezeigt. Wird ein Fehler während der Eingabe erkannt, wird die Karte "Fehler" geöffnet und eine Fehlermeldung mit einer Fehlerbeschreibung angezeigt.

Einzelschrittausführung

Übersicht

Startet man ein Programm über den Menüpunkt "Einzelschritte", erscheint zusätzlich zum Fenster des gestarteten Programms ein weiteres Fenster mit dem Programmtext. Das Programm lässt sich Zeile für Zeile ausführen. Die aktuelle Programmzeile wird im Programmtext blau hinterlegt. Neben der Fehlersuche (engl. "debugging") lässt sich so ein Programm Zeile für Zeile verstehen.



Neben dem Programmtext enthält das Fenster eine Liste der verwendeten Objekte (rechts oben). Der innere Zustand der Objekte lässt sich in dieser Liste durch ein Klick auf das "+"-Symbol anzeigen. Außerdem wird der Zustand der so genannten "virtuellen Maschine" angezeigt (unten rechts), die das Programm ausführt.

Steuerung

Die Steuerung des Programms erfolgt über die Symbolleiste überhalb des Programmtexts:


Es wird eine Zeile des Programms ausgeführt und in der nächsten auszuführenden Zeile (auch innerhalb einer aufgerufenen Methode) angehalten.


Es wird eine Zeile des Programms ausgeführt und in der nächsten Zeile angehalten. Aufgerufene Methoden werden durchlaufen.


Es wird eine Schleife (wiederhole, solange...mache, zähle...mache) oder der Block unterhalb einer wenn...dann-Anweisung ausgeführt und erst in der anschließenden Zeile angehalten. Falls die Anweisung nicht in einer Schleife liegt, wird in der nächsten Zeile angehalten.


Das Programm wird gestoppt und beendet.


Das Programm wird neu gestartet. Dazu wird ein laufendes Programm zunächst automatisch beendet.

Um die Einzelschrittausführung zu beenden, schließen Sie einfach das Fenster und gegebenenfalls offene Fenster von ausgeführten Programmen.

2. Objekte

Objekte erzeugen (📦)

Was sind Objekte?

Objekte sind das Herzstück bei der Programmierung mit KidsPL. Sie stellen Gegenstände wie z.B. ein Bild oder einen Kreis dar. Objekte besitzen in der Regel mehrere Methoden. Methoden erlauben es mit den Objekten etwas Bestimmtes zu machen. So besitzen Bild und Kreis beide eine Methode mit dem Namen 'zeige', die das jeweilige Objekt auf dem Bildschirm anzeigt.

Objekte unterscheiden sich durch Eigenschaften, die man festlegen kann. Bei einem Kreis lässt sich beispielsweise die Farbe bestimmen. Alle Objekte tragen Namen, z.B. könnte man einen roten Kreis 'roterKreis' und eine blauen Kreis 'blauerKreis' nennen. Beide Objekte gehören damit zur gleichen 'Klasse' von Objekten, nämlich 'Kreis', aber haben unterschiedliche Namen und Eigenschaften. Wichtig ist dabei das Folgende: Namen dürfen keine Leerzeichen enthalten. Die Verwendung deutscher Umlaute hingegen erlaubt.

Objekte mit Namen erzeugen

Um Objekte zu erzeugen, schreibt man den frei wählbaren Namen gefolgt von eine Gleichheitszeichen und dem Namen der Klasse gefolgt von einem Klammerpaar '()'. Wenn ein Objekt den Namen 'blauerKreis' erhalten soll und von der Klasse 'Kreis' sein soll, schreibt man also

Programm

```
blauerKreis=Kreis()
```

Das Objekt, genauer der Name des Objekts, wird in blau angezeigt, während der Klassenname in grün angezeigt wird. Das dient zur besseren Unterscheidung. Nach Drücken der ↵-Taste wird das Objekt außerdem im Klassen-Browser (rechts neben dem Programm) angezeigt.

Eigenschaften von Objekten festlegen

Damit das Objekt 'blauerKreis' tatsächlich einen Kreis mit blauer Farbe darstellt, müssen wir beim Erzeugen des Objekts noch die Eigenschaft 'Farbe' des Kreises bestimmen, die "blau" sein soll. Die Eigenschaft 'Farbe' muss man in das Klammerpaar schreiben wiederum gefolgt von einem Gleichheitszeichen und dem Wert "blau".

Programm

```
blauerKreis=Kreis(Farbe="blau")
```

Man nennt die Eigenschaft 'Farbe' alternativ auch Parameter. Es lassen sich häufig auch mehrere solcher Eigenschaften bestimmen. Zum Beispiel kann man beim Erzeugen eines Objektes der Klasse 'Kreis' noch zusätzlich den Parameter 'Größe' bestimmen. Mehrere Parameter lassen sich durch ein Komma trennen, wobei es auf die Reihenfolge nicht ankommt:

Programm

```
meinKreis=Kreis(Farbe="blau", Größe=400)
```

Zahlen schreibt man im Gegensatz zu Texten nicht in Anführungszeichen. KidsPL unterscheidet bei Namen für Objekte, Parameter und Klassen übrigens nicht zwischen Groß- und Kleinschreibung.

Weitere Beispiele

Hier findest du einige weitere Beispiele. Die graue Schrift kennzeichnet Kommentare, die grundsätzlich durch ein Prozentzeichen eingeleitet werden. Kommentare dienen der besseren Lesbarkeit von Programmen und werden vom Computer ignoriert. Man beachte: Es ist prinzipiell auch erlaubt für Objekte Namen zu benutzen, die bereits Klassen bezeichnen.

Programm

```
% Ein Bild, das eine Bilddatei mit dem Namen "Auto" lädt.
% Als Name des Objekts wird 'BildEinesAutos' gewählt:
BildEinesAutos=Bild(Name="Auto")
% Ein weiteres Objekt der Klasse Bild:
Hund=Bild(Name="Hund", Größe=200)
% Ein Objekt der Klasse Rechteck:
MeinRechteck=Rechteck(Farbe="grün", Breite=400, Höhe=50)
% Ein Qudrat. Da keine Eigenschaften gesetzt werden, enthält das
% Quadrat, das wir einfach nur 'Quadrat nennen wollen, die Standardwerte
% Größe=100 und Farbe="rot":
Quadrat=Quadrat()
```

Objekte lassen sich auch interaktiv durch einen Klick auf das Symbol 📦 über einen Dialog erstellen. Alternativ kann dies auch durch einen Klick auf "Neues Objekt" im Klassen-Browser (rechts neben dem Programmtext) geschehen.

Methoden aufrufen (⚙)

Methoden aufrufen

Methoden erlauben es, mit Objekten bestimmte Aktionen auszuführen. Ein Objekt der Klasse 'Bild' lässt sich z.B. mit Hilfe der Methode 'zeige' darstellen. Zunächst muss aber ein Objekt dieser Klasse wie im Abschnitt "Objekte erzeugen" beschrieben erzeugt werden. Der Aufruf einer Methode besteht aus dem Namen der Methode gefolgt von einem Doppelpunkt und dem Namen des Objekts.

Programm

```
Frosch=Bild(Name="Frosch")
zeige:Frosch
```

Methoden mit Parametern

Auch Methoden können mit Parametern aufgerufen werden. Der Methode 'zeige' kann man beispielsweise mitteilen, an welcher Stelle auf dem Bildschirm das Bild dargestellt werden soll. Dies geschieht mit den Parametern 'x' (horizontale Richtung) und 'y' (vertikale Richtung). Der Punkt auf dem Bildschirm mit 'x=0' und 'y=0' befindet sich in der oberen linken Ecke.

Programm

```
Frosch=Bild(Name="Frosch")
zeige(x=100,y=100):Frosch
```

Welche Methoden ein Objekt einer bestimmten Klasse besitzt, lässt sich in der Klassenreferenz erfahren. Es können natürlich auch mehrere Methoden eines Objekts nacheinander aufgerufen werden.

Programm

```
Frosch=Bild(Name="Frosch")
zeige(x=100,y=100):Frosch
nachRechts(Punkte=100):Frosch
```

Anonyme Objekte

Wir haben gelernt, dass beim Erzeugen von Objekten immer ein Name vergeben werden muss. Eine Ausnahme ist aber, wenn wir das neu erzeugte Objekt (ohne Objektnamen und Gleichheitszeichen) direkt hinter einem Methodennamen gefolgt vom Doppelpunkt schreiben. Das Programm

Programm

```
zeige(x=100,y=100):Bild(Name="Frosch")
```

macht das gleiche wie

Programm

```
Frosch=Bild(Name="Frosch")
zeige(x=100,y=100):Frosch
```

Solche Objekte ohne Namen nennt man anonyme Objekte. Es kann wie im vorherigen Beispiel immer nur genau eine Methode eines anonymen Objekts aufgerufen werden.

Methoden mit Rückgabewerten

Methoden können so genannte Rückgabewerte besitzen. Solche Methoden können z.B. Werte für Parameter liefern. Ein Beispiel: Die Klasse 'Farbmixer' erlaubt das Anmischen von eigenen Farben. Die Methode 'gibFarbe' hat als Rückgabewert einen Text, der benutzt werden kann um die Farbe eines Quadrats zu bestimmen:

Programm

```
MeinFarbmixer=Farbmixer(rot=0,grün=100,blau=100) % Mische Türkis an
Quadrat=Quadrat(Farbe=gibFarbe:MeinFarbmixer)
zeige:Quadrat
```

Solche Methoden mit Rückgabewerten geben allgemein ein bestimmtes Objekt zurück und können hinter jedem '=' und ':' stehen.

Auch Methodenaufrufe lassen sich dialoggestützt durch einen Klick auf  einfügen. Im Klassen-Browser kann man einen Methodenaufruf durch einen Doppelklick auf den Objektnamen und anschließend auf den gewüschten Methodennamen im Editor einfügen.

Groß- und Kleinschreibung

Bei der Schreibweise von Namen von Objekten, Methoden, Klassen und Parametern wird nicht zwischen Klein- und Großschreibung unterschieden. So sind beispielsweise die Objektnamen 'kleinerKreis', 'KleinerKreis' und 'KLEINERKREIS' identisch. Es wird aber empfohlen bei einer Schreibweise (möglichst der korrekten Rechtschreibung entsprechend) zu bleiben. Bei zusammengesetzten Namen empfiehlt sich die Großschreibung des zweiten Wortes, dritten Wortes, usw. (z.B. 'großerGrünerKreis'). Deutsche Umlaute und das ß können ohne Einschränkung benutzt werden. Leerzeichen hingegen dürfen nicht in Namen vorkommen.

Text

Text erzeugen und schreiben

Um Objekte von der Klasse 'Text' zu erzeugen, kann die bekannte Schreibweise verwendet werden. Mit dem Parameter 'Wert' kann der zu repräsentierende Text gesetzt werden:

Programm

```
MeinText=Text(Wert="Hallo")
```

Mit der Methode 'schreibe' des 'Text'-Objektes, lässt sich der Text ausgeben:

Programm

```
MeinText=Text(Wert="Hallo")
schreibe:MeinText
```

Diese Methode kennt auch den Parameter 'Farbe', so dass sich Texte auch farbig ausgeben lassen:

Programm

```
Begrüßung=Text(Wert="Willkommen zu diesem Programm!")
schreibe(Farbe="grün"):Begrüßung
```

Kurzform

Da Objekte der Klasse 'Text' sehr häufig verwendet werden, gibt es eine Abkürzung. Statt 'Text (Wert="Hallo")' lässt sich einfach nur "Hallo" schreiben:

Programm

```
MeinText="Hallo"
schreibe:MeinText
```

Die Kurzform sollte immer verwendet werden, da sie deutlich übersichtlicher ist! Noch einfacher lässt sich das Programm schreiben, wenn wir auf einen Namen für den Text verzichten und stattdessen schreiben:

Programm

```
schreibe:"Hallo"
```

Verkettung von Texten

Texte können mit einem Pluszeichen ('+')aneinander gehängt werden. Ein Beispiel für diese so genannte Verkettung von zwei oder mehreren Text-Objekten ist:

Programm

```
Name="Jan"
Wohnort="Berlin"
schreibe:"Meine Name ist " + Name + " und ich wohne in " + Wohnort + "."
```

Zahlen

Zahlen erzeugen

Auch für Zahl-Objekte gibt es eine Kurzform zum Erzeugen von Zahlen. Dabei muss statt

Programm

```
meineZahl=Zahl(Wert=21)
```

einfach nur die Zahl geschrieben werden, nämlich:

Programm

```
meineZahl=21
```

Mit der Methode 'schreibe' lässt sich eine Zahl auf dem Bildschirm ausgeben, d.h.

Programm

```
meineZahl=21
schreibe:meineZahl
```

oder einfach:

Programm

```
schreibe:21
```

Mit Zahlen rechnen

Man kann mit Zahlen wie gewohnt rechnen: Es können die Rechenarten plus ('+'), minus ('-'), mal ('*') und geteilt ('/') benutzt werden. Also zum Beispiel:

Programm

```
% Es lassen sich die gewohnten Zeichen zum Rechnen mit Zahlen verwenden:
schreibe:10+6 % schreibt 16
% Man beachte: Punktrechnung (*,/) geht vor Strichrechnung (+,-) und
% Klammern werden zuerst ausgewertet:
eineZahl=12
ergebnis=4*eineZahl-21/(4+3)*2
schreibe:ergebnis %schreibt 42
```

Außerdem hat ein Zahl-Objekt neben der Methode 'schreibe' noch einige weitere Methoden zum Rechnen, die alternativ benutzt werden können:

Programm

```
z=5
erhöhe:z % erhöht um 1 auf 6
addiere(Zahl=10):z % addiert 10 hinzu
schreibe:z % gibt 16 aus
```

Man beachte, dass für Zahlen manchmal keine richtigen Namen, sondern wie in der Mathematik üblich nur Buchstaben benutzt werden. Weitere Informationen zu den Methoden von Zahl-Objekten finden sich in der Klassenreferenz unter Zahl.

Kommazahlen

Neben ganzen Zahlen gibt auch Zahlen mit Komma. Diese lassen sich mit der Klasse 'Kommazahl' erzeugen. Auch hier gibt es eine Kurzform:

Programm

```
z=12.5
z=12303.21
z=7.0
```

Man muss allerdings beachten, dass man statt des Kommas einen Punkt schreiben muss. Auch mit Kommazahlen kann man wie oben angegeben rechnen. Treten Zahlen und Kommazahlen in einer Rechnung auf, ist das Ergebnis immer eine Kommazahl. Um eine Kommazahl in eine Zahl umzuwandeln, kann man die Methode 'gibZahl' benutzen, die die Kommastellen abschneidet, oder die Methode 'runde', die die Kommazahl korrekt auf- oder abrundet. Die Klasse Kommazahl besitzt viele weitere Methoden, wie zum Beispiel mathematische Funktionen. In der Wissenschaft werden Kommazahlen oft mit Exponenten angegeben, so lässt sich eine Million in der 10er-Exponentenschreibweise darstellen als 'z=1.0*exp:6'. Weitere Informationen zur Klasse 'Kommazahl' finden sich in der Klassenreferenz unter Kommazahl.

Generische Objekte

Sammelbehälter

Häufig benötigt man mehrere Objekte der gleichen Klasse. Statt alle Objekte einzeln zu verwalten, kann man diese Objekte in eine Art Sammelbehälter ablegen. Solche Sammelbehälter sind z.B. Listen und Felder. In diesen Sammelbehältern lassen sich eine bestimmte Anzahl von Objekten hineinlegen und auch wieder herausnehmen. Diese Behälter können immer nur eine bestimmte Klasse von Objekten aufnehmen, so dass beim Erzeugen ("Generieren") eines Behälter die Klasse der Objekte angegeben werden muss. Man bezeichnet die Behälter deswegen als generische Objekte.

Listen

Soll eine Liste von Text-Objekten angelegt werden, so muss bei Erzeugen der Liste ein zusätzlicher Parameter 'Typ' angegeben werden. Nach 'Typ' folgt ein Gleichheitszeichen und der Name einer Klasse:

Programm

```
ErsteListe=Liste(Typ=Text)
```

Achtung: Nach der Klasse folgt kein Klammerpaar '()', da hier kein Objekt erzeugt werden soll, sondern nur die Klasse mitgeteilt wird, die als so genannter "generischer Typ" verwendet wird. In diesem Fall ist der Generische Typ die Klasse 'Text'. Es lässt sich aber genauso gut eine Liste von Bild-Objekten erzeugen, indem man schreibt:

Programm

```
ZweiteListe=Liste(Typ=Bild)
```

Gibt man den Parameter 'Typ' nicht an, so wird standardmäßig eine Liste von Zahl-Objekten erzeugt.

Um einer Liste ein Objekt (am Listenende) hinzuzufügen, kann man die Methode 'fügeHinzu' benutzen. Der Parameter 'Wert' der Methode bestimmt das hinzuzufügende Objekt. Man beachte: Das Objekt muss von der in 'Typ' angegebenen Klasse erzeugt worden sein. Hier sind ein paar Beispiele:

Programm

```
% Eine Liste für Texte:
TextListe=Liste(Typ=Text)
fügeHinzu(Wert="Zahnbürste"):TextListe
fügeHinzu(Wert="Welt"):TextListe
% Eine andere Liste für Zahlen:
ZahlListe=Liste(Typ=Zahl)
fügeHinzu(Wert=42):ZahlListe
% Eine weitere Liste für Kreise:
KreisListe=Liste(Typ=Kreis)
fügeHinzu(Wert= Kreis(Farbe="gelb") ):KreisListe
```

Listen-Objekte haben einige weitere Methoden: Die Methode 'gibGröße' gibt z.B. die Anzahl der in der Liste enthaltenen Objekte als Rückgabewert zurück. Die Methode 'gib' gibt ein Objekt aus der Liste als Rückgabewert zurück, das aber in der Liste verbleibt. Um zu bestimmen welches Objekt aus der Liste zurückgegeben werden soll, muss der Parameter 'Index' angegeben werden. 'Index=1' steht für den ersten Eintrag in der Liste, 'Index=2' für den zweiten und so weiter. Im folgenden Beispiel werden drei Text-Objekte der Liste mit dem Namen 'Geschwister' hinzugefügt. Anschließend wird die Anzahl der in der Liste vorhanden Objekte und das Text-Objekt mit dem 'Index=2' ausgegeben:

Programm

```
Geschwister=Liste(Typ=Text)
fügeHinzu(Wert="Peter"):Geschwister
fügeHinzu(Wert="Anna"):Geschwister
fügeHinzu(Wert="Paul"):Geschwister
schreibe:"Ich habe " + gibGröße:Geschwister + " Geschwister."
schreibe:"Meine Schwester heißt " + gib(Index=2):Geschwister +"."
```

Listen sind sehr flexibel beim Hinzufügen und Entfernen von Objekten und besitzen viele weitere Methoden, die in der Klassenreferenz nachgeschlagen werden können.

Felder

Eine weitere Möglichkeit mehrere Objekte zu verwalten, besteht mit der Klasse 'Feld'. Ein wichtiger Unterschied zu Listen ist, dass beim Erzeugen eines Feldes die Anzahl der Plätze im Feld (d.h. die Länge des Feldes) angegeben werden muss. Die Plätze setzt man mit der Methode 'setze' unter Angabe des Indexes:

Programm

```
Geschwister=Feld(Typ=Text,Länge=3)
setze(Index=1,Wert="Peter"):Geschwister
setze(Index=2,Wert="Anna"):Geschwister
setze(Index=3,Wert="Paul"):Geschwister
schreibe:"Ich habe " + gibLänge:Geschwister + " Geschwister."
schreibe:"Meine Schwester heißt " + gib(Index=2):Geschwister +"."
```

Weitere Informationen finden sich in der Klassenreferenz. Listen und Felder sind besonders nützlich im Zusammenhang mit Programmstrukturen wie 'zähle...mache'.

3. Programmstrukturen

wenn...dann (?)

Bedingte Ausführung von Befehlen

Es kommt häufig vor, dass ein Programm auf den Wert eines Objekts (Eingaben vom Benutzer wie z.B. eine Zahl oder einen Text) in einer bestimmten Weise reagieren soll. Tritt eine bestimmte Situation ein, sollen spezielle Anweisungen ausgeführt werden. Man nennt dies bedingte Ausführung.

Zunächst ein Beispiel: Ein Programm soll das Alter einer Person einlesen und entscheiden, ob diese Person volljährig ist. Zunächst muss über einen Dialog ('Zahldialog') das Alter eingelesen werden, was folgendermaßen geschehen kann:

Programm

```
Dialog=ZahlDialog(Text="Gib dein Alter ein!")
zeige:Dialog
Alter=gibZahl:Dialog
schreibe:"Du bist "+Alter+" Jahre alt."
```

Um abzufragen, ob der Benutzer 18 Jahre oder älter ist, lautet die Bedingung 'Alter>=18'. Um eine 'wenn...dann' Anweisung einzufügen, klickt man auf das ? am linken Fensterrand. Anschließend lässt sich die Bedingung eingeben. In dem grünen Kasten unter der Bedingung schreibt man, was gemacht werden soll, falls die Bedingung zutrifft. In diesem Fall soll "Du bist volljährig." geschrieben werden:

Programm

```
Dialog=ZahlDialog(Text="Gib dein Alter ein!")
zeige:Dialog
Alter=gibZahl:Dialog

wenn Alter>=18 dann
    schreibe:"Du bist volljährig."
```

Soll außerdem auch etwas ausgegeben werden, wenn die Bedingung nicht zutrifft, muss man mit der rechten Maustaste auf 'wenn' klicken. Anschließend wählt man in dem erscheinenden Menü 'sonst' aus und es erscheint unter 'sonst' ein zusätzlicher Kasten, der nur dann ausgeführt wird, falls die Bedingung nicht zutrifft:

Programm

```
Dialog=ZahlDialog(Text="Gib dein Alter ein!")
zeige:Dialog
Alter=gibZahl:Dialog

wenn Alter>=18 dann
```

```
    schreibe:"Du bist volljährig."
sonst
    schreibe:"Du bist in "+(18-Alter)+" Jahr(en) volljährig."
```

Sollen noch weitere Bedingungen abgefragt werden, falls die 'wenn...dann' Bedingung nicht zutrifft, lässt sich durch einen Rechtsklick mit der Maus eine 'oder wenn...dann' Bedingung einfügen:

Programm

```
Dialog=ZahlDialog(Text="Gib dein Alter ein!")
zeige:Dialog
Alter=gibZahl:Dialog

wenn Alter>18 dann
    schreibe:"Du bist volljährig."
oder wenn Alter=18 dann
    schreibe:"Du bist gerade volljährig geworden."
sonst
    schreibe:"Du bist in "+(18-Alter)+" Jahr(en) volljährig."
```

Genauso wie Zahlen lassen sich auch Kommazahlen miteinander vergleichen.

Texte vergleichen

Außer Zahlen kann man unter anderem auch Texte vergleichen. Das folgende Programm vergleicht den Text 'Name="jan"' mit "Jan". Verwendet man ein einfaches Gleichheitszeichen, wird der Text verglichen, ohne dass auf Groß- und Kleinschreibung geachtet wird:

Programm

```
Name="jan"

wenn Name="Jan" dann
    schreibe:"Der Name ist Jan."
```

Dieses Programm schreibt deswegen "Der Name ist Jan.". Verwendet man ein doppeltes Gleichheitszeichen '==', so werden die folgenden Anweisungen nur ausgeführt, falls die Texte (unter Beachtung der Groß- und Kleinschreibung) identisch sind:

Programm

```
Name="jan"

wenn Name=="Jan" dann
    schreibe:"Der Name Jan wurde groß geschrieben."

wenn Name=="jan" dann
    schreibe:"Der Name Jan wurde klein geschrieben."
```

Wahr oder falsch

Manchmal ist es praktisch direkt zu bestimmen, ob eine 'wenn...dann' Anweisung ausgeführt werden soll. Dazu benötigt man ein Objekt, das entweder den Wert wahr oder falsch besitzt. Solche Objekte gehören zur Klasse 'Bool' (nach einem bekannten Mathematiker). Solche Objekte lassen sich erzeugen durch 'Objektname=wahr' oder 'Objektname=falsch' und können durch die 'wenn...dann' Anweisung überprüft werden:

Programm

```
farbig=wahr

wenn farbig dann
    zeige:Kreis(Farbe="rot")
sonst
    zeige:Kreis(Farbe="schwarz")
```

Für 'farbig=wahr' wird also ein roter Kreis gezeigt, während für 'farbig=falsch'

Programm

```
farbig=falsch

wenn farbig dann
    zeige:Kreis(Farbe="rot")
sonst
    zeige:Kreis(Farbe="schwarz")
```

ein schwarzer Kreis dargestellt wird. Alternativ hätte man auch 'wenn farbig=wahr dann' schreiben können.

Nicht, und, oder

Es lassen sich auch mehrere Bedingungen mit Hilfe von 'und' und 'oder' verknüpfen. Die Anweisungen werden bei 'und' nur ausgeführt falls beide Bedingungen (links und rechts vom 'und') erfüllt sind. Bei 'oder' muss hingegen nur eine Bedingung zutreffen. Außerdem lässt sich vor jede Bedingung noch ein

'nicht' stellen, das das Ergebnis der Bedingung umdreht bzw. verneint (aus wahr wird falsch, aus falsch wird wahr). Sind mehrere Verknüpfungen vorhanden, werden zunächst alle 'nicht'-, dann alle 'und'- und danach alle 'oder'-Verknüpfungen ausgeführt. Durch das Setzen von Klammern lässt sich die Reihenfolge der Auswertung bestimmen wie im folgenden etwas komplexeren Beispiel:

Programm

```
z=13
Primzahl=wahr

wenn z>=10 und (nicht Primzahl oder z=13) dann
    schreibe:"Die Zahl ist größer/gleich 10. Sie ist keine Primzahl oder aber gerade 13."
```

Gleichheit anderer Objekte*

Es wurden bisher Objekte der Klassen Zahl, Kommazahl, Text und Bool verglichen. Dabei wurden die in den Objekten gespeicherten Werte verglichen. Andere Objekte (wie z.B. ein Kreis) enthalten hingegen mehr als einen Wert und werden nicht anhand ihrer Werte verglichen. Bei diesen Objekten wird verglichen, ob die Objekte tatsächlich identisch sind. Der Unterschied ist folgender: Zwei Kugelschreiber mögen absolut gleich aussehen, trotzdem sind sie nicht die selben (d.h. identisch).

Wenn z.B. ein Kreis erzeugt wird mit 'Kreis1=Kreis()' so wird immer ein neuer Kreis erzeugt. Der Kreis 'Kreis2=Kreis()' sieht zwar vollkommen gleich aus, ist aber nicht der selbe. Das wird unter anderem deutlich, wenn man nachträglich die Farbe von 'Kreis1' ändert ('setze(Farbe="gelb"):Kreis1'). Nach der Zuweisung 'Kreis3=Kreis1' hingegen, bezeichnen 'Kreis1' und 'Kreis3' hingegen den selben Kreis (die Objekte sind also identisch). Das folgende Programm

Programm

```
Kreis1=Kreis()
Kreis2=Kreis()
Kreis3=Kreis1

wenn Kreis1=Kreis2 dann
    schreibe:"Kreis1 und Kreis2 sind das selbe Objekt."

wenn Kreis1=Kreis3 dann
    schreibe:"Kreis1 und Kreis3 sind das selbe Objekt."
```

gibt deswegen nur "Kreis1 und Kreis3 sind das selbe Objekt." aus.

wiederhole ()

Mehrfache Ausführung von Befehlen

Häufig ist es nötig einen Befehl häufig zu wiederholen. Statt den Befehl entsprechend häufig hinzuschreiben kann z.B. eine 'wiederhole' Struktur (auch Schleife genannt) benutzt werden. Die 'wiederhole' Schleife führt Befehle beliebig oft aus und bricht ohne weiteres erst dann ab, wenn das Programm vom Benutzer beendet wird.

Ein Beispiel: Es soll ein Bild auf dem Bildschirm gezeigt werden und anschließend nach rechts bewegt werden. Dies lässt sich mit der Methode 'nachRechts' des Bild-Objekts realisieren. Damit das Bild nicht zu schnell nach rechts bewegt wird, muss das Programm nach dem Verschieben des Bildes eine Zeit lang warten ('warte:Zeit(Einheiten=5)'). Allerdings wird das Bild so nur einmal nach rechts bewegt und nicht fortwährend. Damit das geschieht, muss eine wiederhole Schleife mit Hilfe des -Knopfes eingefügt werden und die zu wiederholenden Befehle in den Kasten unter 'wiederhole' geschrieben werden. Das Programm sieht dann so aus:

Programm

```
Auto=Bild(Name="Auto")
zeige:Auto

wiederhole

    nachRechts:Auto
    warte:Zeit(Einheiten=5)
```

Eine wiederhole-Schleife könnte zum Beispiel auch dazu benutzt werden, um dem Computer "das Zählen beizubringen". In der Schleife wird 'MeineZahl' zunächst ausgegeben, dann erhöht und dann eine Sekunde gewartet:

Programm

```
MeineZahl=1
wiederhole

    schreibe:MeineZahl
    erhöhe:MeineZahl
    warte:Zeit(Sekunden=1)
```

Es gibt übrigens noch weitere Schleifenstrukturen in KidsPL: die 'solange'-Schleife sowie eine 'zähle'-Schleife.

Abbrechen von Schleifen

Schleifen lassen sich mit dem Befehl 'abbruch' allerdings auch abbrechen, so dass die Befehle nicht endlos wiederholt werden. Dies geschieht in der Regel durch eine 'wenn...dann' Anweisung. Um zum Beispiel nur die Zahlen bis 20 auszugeben, muss man das vorherige Beispiel folgendermaßen verändern:

Programm

```
MeineZahl=1
wiederhole

    schreibe:MeineZahl
    wenn MeineZahl=20 dann

        abbruch

    erhöhe:MeineZahl
    warte:Zeit(Sekunden=1)
```

Es gibt noch einen weiteren Befehl, der die Ausführung von Schleifen beeinflusst: Die 'weiter' Anweisung führt dazu, dass die restlichen Befehle innerhalb eine Schleife nicht ausgeführt werden und das Programm mit dem ersten Befehl der Schleife fortsetzt. Eine 'weiter' Anweisung ließe sich zum Beispiel dazu benutzen nur die geraden Zahlen auszugeben. Dazu wird überprüft, ob die Zahl durch zwei teilbar ist. Falls nicht, wird die Zahl erhöht und durch den Befehl 'weiter' wieder beim Beginn der Schleifen angefangen. Falls ja, wird die Zahl ausgegeben und erhöht:

Programm

```
MeineZahl=1
wiederhole

  wenn nicht teilt(Teiler=2):MeineZahl dann

    erhöhe:MeineZahl
    weiter

  schreibe:MeineZahl
  erhöhe:MeineZahl
  warte:Zeit(Sekunden=1)
```

Ein weiteres Beispiel

Folgendes Programm benutzt die 'wiederhole' Anweisung, um viele unterschiedliche bunte Kreise auf dem Bildschirm anzuzeigen. Die Parameter für den Kreis werden dabei zufällig bestimmt:

Programm

```
% Male viele bunte Kreise:
wiederhole

  % Benutze Zufallszahlen um Ort und Größe des Kreises zu bestimmen:
  ZufallsX=gib:Zufallszahl(kleinsteZahl=1,größteZahl=800)
  ZufallsY=gib:Zufallszahl(kleinsteZahl=1,größteZahl=600)
  Zufallsgröße=gib:Zufallszahl(kleinsteZahl=1,größteZahl=200)
  % Ohne Parameter liefert Farbmixer() ein zufällige Farbe:
  Zufallsfarbe=gibFarbe:Farbmixer()
  % Zeige einen Kreis auf dem Bildschirm:
  zeige(x=ZufallsX,y=ZufallsY):Kreis(Größe=Zufallsgröße,Farbe=Zufallsfarbe)
  % Warte eine Sekunde bis der nächste Kreis gezeichnet wird:
  warte:Zeit(Sekunden=1)
```

solange...mache (?)

Bedingte wiederholte Ausführung von Befehlen

Eine 'solange...mache' Struktur wiederholt Anweisungen so häufig, solange eine Bedingung erfüllt ist. Ist die Bedingung nicht (mehr) erfüllt, wird diese Schleife abgebrochen, anderenfalls werden die Befehle im Kasten unter der 'solange...mache' Anweisung ausgeführt und anschließend wieder die Bedingung geprüft usw. Die 'solange...mache' Schleife lässt sich durch Klicken auf den ?-Knopf einfügen.

Ein Beispiel: Um die Zahlen von 1 bis 20 auszugeben muss die Zahl 'MeineZahl' in der Schleife ausgegeben und anschließend erhöht werden. Die Schleife soll ausgeführt werden, solange 'MeineZahl<=20' ist:

Programm

```
MeineZahl=1
solange MeineZahl<=20 mache
    schreibe:MeineZahl
    erhöhe:MeineZahl
```

Ein weiteres Beispiel: Möchten wir ein Bild zeigen und anschließend schrittweise nach rechts verschieben, lässt sich die Methode 'nachRechts' des Bild-Objekts verwenden. Die horizontale Position des Bildes gibt die Methode 'gibX' zurück. Am rechten Fensterrand erreicht das Bild die horizontale Position 800. Soll das Programm beim Erreichen des rechten Fensterrandes stoppen, so sieht das Programm wie folgt aus:

Programm

```
Auto=Bild(Name="Auto")
zeige:Auto

solange gibX:Auto<800 mache
    nachRechts:Auto
    warte:Zeit(Einheiten=5)
```

solange bis...mache

Wird in der 'solange...mache' Anweisung vor der Bedingung ein 'bis' geschrieben, wird die Schleife abgebrochen, *falls* die Bedingung erfüllt ist. Die Schleife wird dann so oft ausgeführt, solange die Bedingung *nicht* zutrifft. Das letzte Programm ließe sich also mit Hilfe von 'solange bis...mache' schreiben als:

Programm

```
Auto=Bild(Name="Auto")
zeige:Auto

solange bis gibX:Auto=800 mache
    nachRechts:Auto
    warte:Zeit(Einheiten=5)
```

Abbrechen von Schleifen

'Solange...mache'-Scheifen lassen sich mit dem Befehl 'abbruch' abbrechen und mit dem Befehl 'weiter' lassen sich die restlichen Anweisungen einer Schleife überspringen. Beispiele finden sich unter "Abbrechen von Schleifen" unter wiederhole.

zähle...mache ($_{1,}^{2,}{}^{3,}$)

Zählschleifen

Mit Hilfe des Befehls 'zähle bis ... mache', der sich durch ein Klick auf das $_{1,}^{2,}{}^{3,}$-Symbol einfügen lässt, lassen sich Anweisungen wiederholen. Im einfachsten Fall zählt der Computer von eins bis zu einer

angegebenen Zahl und führt nach jedem hochzählen den Inhalt der Schleife aus. Das folgende Programm schreibt z.B. zehn mal "Hallo", indem es von eins bis zehn zählt und jedes mal "Hallo" ausgibt:

Programm

```
zähle bis 10 mache
    schreibe:"Hallo"
```

Durch ein zusätzliches 'von ...' lässt sich die Zahl bestimmen, bei der begonnen werden soll zu zählen:

Programm

```
zähle von 11 bis 13 mache
    schreibe:"Hallo"
```

Häufig möchte man den Zählerstand explizit benutzen: Dazu lässt sich nach 'zähle' der Name einer Zahl angeben, in die der Zählerstand geschrieben werden soll. Das folgende Programm gibt den Zählerstand beim Durchlaufen der Schleife aus:

Programm

```
zähle Zähler von 11 bis 13 mache
    schreibe:"Zählerstand: "+Zähler
```

Ausgegeben wird folglich: "Zählerstand: 11", "Zählerstand: 12" und "Zählerstand: 13". Normalerweise wird bei der 'zähle...mache' Anweisung der Zähler um plus eins erhöht. Die Schrittweite lässt sich aber verändern, indem zusätzlich z.B. '+2', '+3', '+4' usw. beziehungsweise '-1', '-2', usw. angegeben wird. Die Ausgabe des nächsten Programms ist entsprechend: "Zählerstand: 11", "Zählerstand: 13", "Zählerstand: 15","Zählerstand: 17".

Programm

```
zähle Zähler +2 von 11 bis 17 mache
    schreibe:"Zählerstand: "+Zähler
```

Die 'zähle bis ... mache' Schleife wird abgebrochen, wenn die Zahl beim Erhöhen (bzw. beim Erniedrigen, falls die Schrittweite negativ ist) größer (bzw. kleiner) als die nach 'bis' genannte Zahl wird.

Beispiel

Im folgenden Beispiel befindet sich ein Adler im Sturzflug! Eine Zahl 'Zahl' wird dabei von 100 bis 900 gezählt. Diese Zahl bestimmt die horizontale Position ('x') des Adlers, während die vertikale Position ('y') nur um eine Drittel dieser Zahl größer werden soll:

Programm

```
Adler=Bild(Name="Adler")
Zahl=100
Ende=900

zähle Zahl bis Ende mache

    zeige(x=Zahl,y=Zahl/3):Adler
    warte:Zeit(Einheiten=8)
```

Man beachte, dass das Zahl-Objekt auch schon vorher im Programm benutzt werden kann. Fehlt ein 'von ...', so behält die Zahl ihren Wert bei. Außerdem lässt sich die Zahl theoretisch auch innerhalb der Schleife verändern (z.B. um bei einer bestimmten Bedingung noch einmal von vorne zu beginnen).

4. Klassen und Methoden

Klassen (📦)

Benutzerdefinierte Klassen

Bisher wurden Objekte von Klassen erzeugt, die vorgegeben waren wie z.B. Kreis, Quadrat, Zahl, usw. Klassen können aber auch selbst geschrieben werden. Dadurch können Objekte erzeugt werden, die selbst formulierte Aufgaben erfüllen können. Als Beispiel soll eine Klasse geschrieben werden, dessen Objekte ein Quadrat mit einem innenliegenden Kreis zeichnen können. Zunächst muss dazu mit dem 📦-Schalter eine neue Klasse eingefügt werden. Als Name für die Klasse soll 'KreisQuadrat' gewählt werden. In den roten Kasten unter 'KreisQuadrat' lassen sich wie auch im Kasten unter 'Programm' beliebig viele Anweisungen schreiben. Dort soll zunächst ein Quadrat- und ein Kreis-Objekt erzeugt werden.

Programm

```
```

KreisQuadrat

```
Quadrat=Quadrat()
Kreis=Kreis()
```

Um unter 'Programm' ein Objekt von der Klasse 'KreisQuadrat' zu erzeugen, schreibt man 'Objekt=KreisQuadrat()', d.h.:

Programm

```
Objekt=KreisQuadrat()
```

KreisQuadrat

```
Quadrat=Quadrat()
Kreis=Kreis()
```

Wird ein Objekt von 'KreisQuadrat' erzeugt, so werden alle Anweisungen unter 'KreisQuadrat' der Reihe nach ausgeführt. In unserem Fall wird dann ein Objekt 'Quadrat' und ein Objekt 'Kreis' erzeugt. Wichtig ist, dass diese Objekte nicht innerhalb des Kastens unter 'Programm' sichtbar sind. Das bedeutet, dass eine 'zeige:Kreis' im Programm zu einem Fehler führt, da auf das Objekt 'Kreis' nicht zugegriffen werden kann. Unter 'KreisQuadrat' lässt sich aber auf die beiden Objekte zugreifen, so dass sich die beiden Objekte wie folgt zeigen lassen:

Programm

```
Objekt=KreisQuadrat()
```

KreisQuadrat

```
Quadrat=Quadrat()
Kreis=Kreis()
zeige:Quadrat
zeige:Kreis
```

Parameter

Wir sind gewohnt beim Erzeugen eines Objekts Parameter zu bestimmen, die z.B. angeben wie das Objekt aussehen soll. Für 'KreisQuadrat' wäre ein Parameter 'Größe' sinnvoll, der einen Zahlenwert beinhaltet, einen Text-Parameter namens 'FarbeKreis' für die Farbe des Kreises und einen Text namens 'FarbeQuadrat' für die Farbe des Quadrats. Im Programm könnte dann ein 'KreisQuadrat'-Objekt mit 'Objekt=KreisQuadrat(Größe=100, FarbeKreis="gelb", FarbeQuadrat="braun")' erzeugt werden:

Programm

```
Objekt=KreisQuadrat(Größe=100,FarbeKreis="gelb",FarbeQuadrat="braun")
```

Damit dies möglich ist, muss ein Klammerpaar hinter 'KreisQuadrat' gesetzt werden, in das die Parameter geschrieben werden zusammen mit einem Vorgabewert:

KreisQuadrat (Größe=100, FarbeKreis="rot",FarbeQuadrat="orange")

```
Quadrat=Quadrat()
Kreis=Kreis()
zeige:Quadrat
zeige:Kreis
```

Wird ein Objekt nicht unter Angaben aller drei Parameter erzeugt, so wird für die restlichen Parameter der Vorgabewert benutzt. Die Parameter können innerhalb der Klasse 'KreisQuadrat' als normale Objekte benutzt werden und erscheinen entsprechend auch im Klassen-Browser. Für das Beispiel benutzen wir nun die Parameter um dem Kreis und dem Quadrat die richtige Farbe und die richtige Größe zu geben:

Programm

```
Objekt=KreisQuadrat()
```

KreisQuadrat (Größe=100, FarbeKreis="rot",FarbeQuadrat="orange")

```
Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)
zeige:Quadrat
zeige:Kreis
```

Da hierbei beim Erzeugen des Objekts 'Größe' nicht angeben wurde, wird der Vorgabewert '100' benutzt. Man beachte, dass ein Vorgabewert immer angegeben werden muss (auch bei komplexeren Objekten wie z.B. bei Bildern, Rechtecken, Listen, etc.). Möglichkeiten für Vorgabewerte sind z.B. 'EineZahl=12', 'EinText=""', 'EinText=Text()', 'Durchmesser=Zahl()', 'Bild=Bild(Name="Auto")' und 'Namensliste=Liste(Typ=Text)'.

Die Klasse 'Programm'

Es ist bestimmt schon aufgefallen, dass die Klasse 'KreisQuadrat' genauso aussieht wie 'Programm' mit dem darunterstehenden Kasten. Tatsächlich ist 'Programm' auch eine Klasse. Beim Ausführen eines

KidsPL-Programms wird zunächst geschaut, ob es eine Klasse namens 'Programm' gibt. Falls ja, wird ein Objekt der Klasse 'Programm' erzeugt, was dazu führt, dass alles im Kasten unter 'Programm' ausgeführt wird. Falls nicht, wird eine Fehlermeldung erzeugt.

Wichtig ist, dass auch benutzerdefinierte Klassen Methoden besitzen können, was unter Methoden weiter erläutert wird.

Methoden ()

Methoden in Klassen einfügen

Die wesentliche Funktionalität von Klassen wird durch die Methoden der Klassen bestimmt. Eine Methode weist ein Objekt einer Klasse an, etwas Bestimmtes zu tun. Die Klasse 'Kreis' besitzt zum Beispiel eine Methode 'zeige', die den Kreis auf dem Bildschirm darstellt. Mit Hilfe des  -Schalters lässt sich in eine benutzerdefinierte Klasse eine Methoden einfügen.

Als Beispiel wird von der Klasse 'KreisQuadrat' ausgegangen, die ein Quadrat mit einem inneren Kreis zeichnen soll (siehe Abschnitt "Klassen"):

Programm

```
Objekt=KreisQuadrat()
```

KreisQuadrat (Größe=100, FarbeKreis="rot", FarbeQuadrat="orange")

```
Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)
```

Objekte dieser Klasse sollen mit der Methode 'zeige' auf dem Bildschirm angezeigt werden. Nach dem Einfügen der Methode 'zeige' lassen sich mit 'zeige:Quadrat' und 'zeige:Kreis' die beiden Grafikobjekte darstellen:

Programm

```
Objekt=KreisQuadrat()
zeige:Objekt
```

KreisQuadrat (Größe=100, FarbeKreis="rot", FarbeQuadrat="orange")

```
Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)
```

Methode zeige

```
zeige:Quadrat
zeige:Kreis
```

Wichtig ist, dass man innerhalb der Methoden auch auf Objekte zugreifen kann, die in der Klasse selbst erzeugt wurden ('Quadrat' und 'Kreis') und auf die Parameter der Klasse ('Größe', 'FarbeKreis', 'FarbeQuadrat'). Normalerweise werden (z.B. bei der Klasse 'Kreis') der Methode 'zeige' die Parameter 'x' und 'y' übergeben, die bestimmen, wo das Objekt gezeichnet werden soll. Deswegen soll nach dem Namen der Methode noch ein Klammernpaar mit den Namen (und Vorgabewerten) der Parameter stehen. Dies passiert auf die gleiche Weise wie bei den Parametern für die Klasse:

Programm

```
Objekt=KreisQuadrat()
zeige(x=100):Objekt
```

```
KreisQuadrat (Größe=100, FarbeKreis="rot", FarbeQuadrat="orange")

    Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
    Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)

    Methode zeige (x=400,y=300)

        zeige(x=x,y=y):Quadrat
        zeige(x=x,y=y):Kreis
```

Als Standardwerte für 'x' wurde 400 und für 'y' wurde 300 gewählt, also der Mittelpunkt des Fensters.

Methoden mit Rückgabewerten

Einige Methoden sollen Werte zurückgeben: zum Beispiel eine Methode 'gibX', die die x-Position des 'KreisQuadrat'-Objekts zurückgeben soll. Dafür schreibt man nach den Parametern der Methode, falls welche vorhanden sind, ein Gleichheitszeichen gefolgt von einem Vorgabewert für die Rückgabe. Oft wird der Wert in jedem Fall innerhalb der Methode gesetzt, so dass man statt z.B. einer Zahl einfach ein Standard-Zahl-Objekt mit 'Zahl()' erzeugt, um keine Zahl explizit angeben zu müssen. Innerhalb der Methode kann man an beliebiger Stelle den Rückgabewert setzen, indem man den Namen der Methode gefolgt von einem Gleichheitszeichen und einem Objekt, das zugewiesen werden soll, schreibt: Also zum Beispiel 'gibX=400'. Doch wie lässt sich der Wert nun die tatsächliche x-Position zurückgeben? Auf Objekte, die in einer anderen Methode (in diesem Falle 'zeige') erzeugt wurden oder als Parameter mitgegeben wurden, kann nicht zugegriffen werden. Deswegen funktioniert das folgende Beispiel *nicht*:

```
KreisQuadrat (Größe=100, FarbeKreis="rot",FarbeQuadrat="orange")

    Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
    Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)

    Methode zeige (x=400, y=300)

        zeige(x=x,y=y):Quadrat
        zeige(x=x,y=y):Kreis

    Methode gibX = Zahl()

        gibX=x % Fehler, denn auf x kann nicht zugegriffen werden
```

Die Objekte 'x' und auch 'y' müssen schon in der Klasse selber erzeugt werden, damit alle Methoden auf die beiden Objekte zugreifen können. Dazu schreibt man bereits in die Klasse 'x=400' und 'y=300'. Wenn bei der Methode 'zeige' 'x' oder 'y' nicht übergeben wird, sollen die Werte der klassenweiten Objekte natürlich auch benutzt und verändert werden. Verwendet man für 'zeige' die Parameter mit den Namen 'x' und 'y' werden durch das Setzen der Parameter die beiden Objekte verändert. Als Vorgabewerte sollen aber nicht 400 und 300 dienen, sondern der Wert der beiden Objekte selber: Also verwendet man als Parameter 'x=x' und 'y=y'. Die Rückgabe der x-Position in der Methode 'gibX' kann jetzt einfach mit 'gibX=x' erfolgen:

Programm

```
Objekt=KreisQuadrat()
zeige(x=100):Objekt
schreibe:gibX:Objekt
```

```
KreisQuadrat (Größe=100, FarbeKreis="rot",FarbeQuadrat="orange")

    Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
    Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)
    x=400
    y=300
```

```
Methode zeige (x=x, y=y)
    zeige(x=x,y=y):Quadrat
    zeige(x=x,y=y):Kreis
Methode gibX = Zahl()
    gibX=x
```

Noch kürzer geht es indem 'x' bereits als Vorgabewert für die Rückgabe benutzt wird, dann lässt sich auf ein explizites setzen von 'gibX' gänzlich verzichten:

```
Methode gibX = x
```

Eigene Methoden aufrufen

Es lassen sich auch Methoden innerhalb der gleichen Klassen aufrufen und das nicht nur innerhalb der Klasse selber, sondern auch innerhalb der Methoden der Klasse. Normalerweise werden Methoden aufgerufen, indem der Methodenname und anschließend nach einem Doppelpunkt der Objektname geschrieben wird. Innerhalb einer Klasse weiß man natürlich nicht wie ein Objekt der Klasse einmal heißen wird und außerdem können schließlich von einer Klasse mehrere Objekte erzeugt werden! Andererseits weiß das laufende Programm, um welches Objekt es sich gerade handelt, da entweder ein neues Objekt erzeugt oder eine Methode aufgerufen worden ist. Folglich werden Methoden innerhalb einer Klasse (d.h. des selben Objekts) nur mit dem Methodennamen (ohne folgenden Doppelpunkt) aufgerufen. Soll z.B. beim Erzeugen eines Objekts der Klasse 'KreisQuadrat' das Objekt sofort angezeigt werden, lässt sich der Aufruf der Methode 'zeige' wie folgt einfügen:

Programm

```
Objekt=KreisQuadrat()
KreisQuadrat (Größe=100,FarbeKreis="rot",FarbeQuadrat="orange")
    Quadrat=Quadrat(Farbe=FarbeQuadrat,Größe=Größe)
    Kreis=Kreis(Farbe=FarbeKreis,Größe=Größe)
    x=400
    y=300
    zeige

    Methode zeige(x=400, y=300)
        zeige(x=x,y=y):Quadrat
        zeige(x=x,y=y):Kreis
        schreibe:"Position x: "+gibX+", y: "+gibY
    Methode gibX = x

    Methode gibY = y
```

Methodenaufrufe in der Klasse selbst müssen nach dem Erzeugen aller Objekte stehen, die man in den Methoden verwenden möchte. Außerdem wurde in diesem Beispiel die Methode 'zeige' so verändert,

dass sie jedes Mal die Position des Objekts anzeigt. Die Methode benutzt dazu die Methoden 'gibX' und 'gibY'. Parameter lassen sich wie gewohnt übergeben, so wäre z.B. auch der Methodenaufruf 'zeige (x=100,y=100)' zulässig.

Klassen erweitern (Vererbung)*

Klassen erweitern

Eine der wichtigsten Konzepte der so genannten objektorientierten Programmierung ist die Möglichkeit bereits vorhandene Klassen zu erweitern. Statt eine Klasse völlig neu zu programmieren, kann man auf Klassen zurückgreifen, die schon einen Teil der gewünschten Funktionalität bereit stellen. Anders ausgedrückt lässt sich eine vorhandene Klasse spezialisieren oder "ableiten". Die abgeleitete Klasse "erbt" die Methoden der vorhandenen Klasse, der so genannten "Elternklasse", und kann eigene Methoden hinzufügen.

Als Beispiel soll die Klasse 'Kreis' so erweitert werden, dass statt der 'Größe' des Kreises (also des Durchmessers) der Parameter 'Radius' verwendet werden soll. Außerdem sollen sich die Koordinaten der Methode 'zeige' auf die obere linke Ecke und nicht auf den Mittelpunkt des Kreises beziehen. Ohne die Vererbung bliebe nichts weiter übrig als eine Klasse 'RadiusKreis' zu schreiben mit *allen* Methoden. Eine bessere Möglichkeit bietet die Erweiterung der vorhandenen Klasse 'Kreis'. Um eine Klasse zu erweitern schreibt man nach dem gewünschten Namen der (spezialisierten) Klasse das Schlüsselwort 'erweitert' gefolgt von dem Namen der Elternklasse 'Kreis':

Programm

```

```

RadiusKreis erweitert Kreis

```

```

Objekte von der Klasse 'RadiusKreis' lassen sich wie gewohnt erzeugen. Da 'RadiusKreis' die Methoden von der Klasse 'Kreis' erbt, lässt sich auch wie gewohnt die Methode 'zeige' aufrufen:

Programm

```
MeinKreis=RadiusKreis()
zeige(x=0,y=0):MeinKreis
```

RadiusKreis erweitert Kreis

```

```

Im Klassen-Browser erscheint unter der Klasse 'RadiusKreis' Liste der "geerbten" Methoden der Elternklasse 'Kreis'. Man beachte, dass in diesem Fall keine Parameter wie 'Größe' oder 'Farbe' beim Erzeugen des Kreises erlaubt sind. Diese Parameter müssen (wie gewohnt) nach dem Namen der Klasse angegeben werden. Für Klasse 'RadiusKreis' sollte es einen Paramter 'Radius' geben und außerdem sollte man die 'Farbe' des Kreises bestimmen können! Diese Paramter können (müssen aber nicht) benutzt werden, um die Paramter der Elternklasse 'Kreis' zu bestimmen. In unserem Fall setzen wir den Parameter der Elternklasse 'Größe' gleich dem doppeltem Wert des Parameters 'Radius' und "leiten" den Parameter 'Farbe' zur Elternklasse 'Kreis' weiter. Objekte der Klasse 'RadiusKreis' lassen sich nun folglich mit den Paramtern 'Radius' und 'Farbe' erzeugen.

Programm

```
MeinKreis=RadiusKreis(Radius=200)
zeige(x=0,y=0):MeinKreis
```

RadiusKreis(Radius=50,Farbe="grün") **erweitert** Kreis(Größe=2*Radius,Farbe=Farbe)

```

```

In die Klasse 'RadiusKreis' lassen sich jetzt weitere Methoden einfügen (wie zum Beispiel eine Methode 'gibRadius'), die die Klasse 'Kreis' um neue Funktionalitäten erweitert:

Programm

```
MeinKreis=RadiusKreis(Radius=200)
zeige(x=0,y=0):MeinKreis
schreibe:"Der Radius von MeinKreis ist "+ gibRadius:MeinKreis +"."
```

RadiusKreis(Radius=50,Farbe="grün") **erweitert** Kreis(Größe=2*Radius,Farbe=Farbe)

```
schreibe:"Radius: "+Radius
schreibe:"Durchmesser: "+(2*Radius)
schreibe:"Farbe: "+Farbe

Methode gibRadius=Radius
    
```

Als letztes soll die Klasse so verändert werden, dass sich die Koordinaten der Methode 'zeige' auf die obere linke Ecke und nicht mehr auf den Mittelpunkt des Kreises beziehen. Es lässt sich die Methode 'zeige' der Elternklasse 'Kreis' zwar nicht direkt ändern, wohl aber durch eine eigene Methode 'zeige' "überdecken" die sich in der Klasse 'RadiusKreis' befindet. Wird im Programm 'zeige:MeinKreis' aufgerufen, so wird dann die Methode 'zeige' der Klasse 'RadiusKreis' und nicht mehr der Klasse 'Kreis' ausgeführt. Das gilt auch, falls die Methode durch z.B. 'zeige(x=x+Radius,y=y+Radius)' innerhalb der Klasse 'RadiusKreis' aufgerufen wird.

Da aber die Methode 'zeige' der Elternklasse zum Zeichnen des Kreises benötigt wird, stellt uns das vor ein Problem. Abhilfe schafft der '~'-Operator, der vor den Methodennamen gestellt die entsprechende Methode der Elternklasse aufruft. Mit der Anweisung '~zeige' wird folglich die Methode 'zeige' der Klasse 'Kreis' aufgerufen. Vervollständigt sieht das Programm dann folgendermaßen aus:

Programm

```
MeinKreis=RadiusKreis(Radius=200,Farbe="orange")
zeige(x=0,y=0):MeinKreis
```

RadiusKreis(Radius=50,Farbe="grün") **erweitert** Kreis(Größe=2*Radius,Farbe=Farbe)

```
%Beziehe Koordinaten of obere linke Ecke des Kreises:
x=400-Radius
y=300-Radius

Methode gibRadius=Radius
    

Methode zeige(x=x,y=y)
    %Beziehe Koordinaten of obere linke Ecke des Kreises:
    ~zeige(x=x+Radius,y=y+Radius)
```

Wird ein '~'-Zeichen vor einen Methodennamen gestellt, so muss die Methode in der Elternklasse (oder in der Elternklasse der Elternklasse usw.) existieren. Existiert die Methode nur in der abgeleiteten Klasse, so kommt es zu einer Fehlermeldung. Außerdem ist die Verwendung von außerhalb, also zum Beispiel '~zeige:MeinKreis' in der Klasse 'Programm', unzulässig.

Virtuelle Methoden und Benutzereingaben

Es ist nun bekannt wie ein abgeleitete Klasse Methoden ihrer Elternklasse aufruft, doch wie kann eine Elternklasse eine Methode der abgeleiteten Klasse aufrufen und ist das überhabt sinnvoll? Betrachten wir die Klasse 'Fenster', die ein Fenster anzeigen und wieder entfernen sowie auf Benutzereingaben wie Maus- und Tastaturaktionen reagieren kann. Beim Start eines KidsPL-Programms wird sogar standardmäßig ein Objekt der Klasse 'Fenster' erzeugt und die Methode 'zeige' aufgerufen. Deswegen erscheint ein KidsPL-Programm immer in einem Fenster. Die einzige Ausnahme von der Regel tritt ein, wenn die Klasse 'Programm' von der Klasse 'Fenster' abgeleitet (erweitert) wird. In diesem Fall kann man insbesondere selbst steuern, ob und wann ein Fenster angezeigt werden soll.

Die Klasse 'Fenster' ruft zum Beispiel immer, wenn eine Maustaste gedrückt wird, ihre Methode 'wennMausGedrückt' auf. Diese Methode ist aber keine normale Methode, sondern ist 'virtuell'. Das bedeutet, dass eine Methode aus der abgeleiteten Klasse mit dem selben Namen die Methode nicht nur "überdeckt" sondern "überschreibt". Das hat den Effekt, dass die Methode 'wennMausGedrückt' der abgeleiteten Klasse, falls diese existiert, aufgerufen wird, falls eine Maustaste gedrückt wird. Falls die Methode nicht virtuell wäre, würde immer die Methode 'wennMausGedrückt' der Elternklasse und nicht die Methode der abgeleiteten Klasse 'Fenster' aufgerufen (auch wenn die Methode der abgeleiteten Klasse nach außen hin die Methode der Elternklasse "überdeckt"). Parameter der virtuelle Methode können (müssen aber nicht) benutzt und um eigene Paramter ergänzt werden. Die Methode 'wennMausGedrückt' übergibt zum Beispiel die Zahl-Paramter 'x', 'y' und 'Taste'. Ein Programm, dass an jeder Stelle des Fensters, an der eine Maustaste gedrückt wird, einen Kreis zeichnet, sieht zum Beispiel folgendermaßen aus:

```
Programm erweitert Fenster

    zeige(Titel="Mein Programm")

    Methode wennMausGedrückt (x=0,y=0)

        zeige(x=x,y=y):Kreis(Größe=20)
```

Wie man auf andere Benutzereingaben reagieren kann, lässt sich der Klassenreferenz zu Fenster entnehmen.

Konzepte, die KidsPL nicht unterstützt*

Zuletzt soll noch auf Sprachkonzepte eingegangen werden, die es in anderen (objektorientierten) Sprachen gibt, aber die von KidsPL nicht unterstützt werden. Auf diese Konzepte wurde beim Sprachentwurf explizit verzichtet, da sie KidsPL deutlich komplizierter gemacht hätten.

Einfache Datentypen

Der Vorteil einfacher Datentypen liegt insbesondere im Geschwindigkeitsvorteil gegenüber der Verarbeitung von Objekten. Damit nicht von Anfang an zwei widersprüchliche Konzepte erlernt werden müssen, nämlich einfache und objektorientierte Typen, gibt es in KidsPL ausschließlich objektorientierte Typen. Wie aber schon bekannt ist, gibt es für die Objekte von den Klassen Zahl, Text, Kommazahl und Bool eine vereinfachende Syntax zum Erzeugen dieser Objekte als auch zusätzliche Operatoren.

Deklarationen

Es können in KidsPL keine reinen Deklarationen vorgenommen werden, d.h. Anweisungen, die einem Bezeichner einen Typ, aber keinen Wert zuweisen. Trotzdem ist KidsPL eine vollständig typensichere Sprache: Die Deklaration in KidsPL geschieht automatisch, wenn einem Bezeichner ein Wert zugewiesen wird. Bezeichner bleiben an den Typ dieses Wertes gebunden. In Funktionsköpfen werden in der gleichen Weise durch die Zuweisung von Vorgabewerten die Typen der Parameter bestimmt.

Zeiger

Da in KidsPL keine Deklarationen ohne Definition erlaubt sind, können in KidsPL keine Laufzeitfehler durch leere Referenzen (so genannte "null pointer exceptions") auftreten. Es gibt in KidsPL aus vielerlei

Gründen keine Möglichkeit die Speicheradressen von Objekten, so genannte Zeiger (engl. pointer), zu manipulieren.

Erweiterte objektorientierte Konzepte

In KidsPL gibt es keine Mehrfachvererbung, keine Interfaces und keinen inneren und sogenannte Adapter-Klassen. Außerdem können keine Operatoren definiert werden. Diese Konzepte hätten der Sprache unnötige Komplexität verliehen und sie dadurch weniger leicht verständlich gemacht.

Typenumwandlung und Polymorphie

Es gibt in KidsPL keine Möglichkeit einen Typ (d.h. eine Klasse) in einen anderen umzuwandeln (casting). Eine Umwandlung von z.B. Zahlen zu Text ist aber indirekt über die Methoden der Objekte möglich. Zusätzlich nehmen auch einige Operatoren (+,-,*,/) solche Umwandlungen vor. Des Weiteren gibt es auch keine Möglichkeit Objekte an den Typ einer ihrer Elternkassen zu binden (Polymorphie). Letzteres Konzept bildet zwar ein mächtiges Werkzeug, gehört aber auch zu den komplexeren Methodiken der objektorientierten Programmierung und wird erfahrungsgemäß zunächst als verwirrend empfunden, da der Typenbegriff dadurch stark erweitert wird. Nach Abwägung der Vor- und Nachteile wurde auf Polymorphie in KidsPL verzichtet.