PROGRAMMER'S CHOICE

Ralph Steyer

JavaFX

Dynamische und interaktive Java-Applikationen mit JavaFX

ADDISON-WESLEY

Eclipse,
NetBeans,
Java SDK

CD

3 Das Syntaxkonzept von JavaFX Script

Sie haben nun alle Voraussetzungen zur Erstellung und Ausführung von JavaFX-Applikationen geschaffen und auch schon erste Beispiele gesehen. Bezüglich der Syntax und des Konzeptes von JavaFX sollten aber bei diesen ersten Beispielen zahlreiche Fragen offen geblieben sein. In diesem Kapitel gehen wir nun genauer auf das syntaktische Konzept von JavaFX als Skriptsprache ein.

JavaFX Script bezeichnet eine so genannte **deklarative** und **statisch typisierte** Programmiersprache respektive Skriptsprache[58]. Es gibt in einer solchen Sprache z. B. Techniken wie **Funktionen**, **Klassen**, **Deklarationen**, **inkrementelle Evaluierung** etc.

Aber JavaFX Script hat zudem einige Besonderheiten aufzuweisen. Mit dieser neuen Skriptsprache von Sun können Sie direkte Aufrufe auf alle Java-APIs durchführen, die auf einer Plattform verfügbar sind. Das unterscheidet sie beispielsweise von bisher schon vorhandenen Konkurrenzsprachen wie JavaScript[59]. Aber obwohl JavaFX auf Java basiert, unterscheidet sich die Sprache dennoch in mehreren zentralen Details massiv von der streng objektorientierten Verwandtschaft.

58 Keine Angst, wenn Sie diese Fachbegriffe (noch) nicht verstehen – die Erklärungen sind ja expliziter Bestandteil dieses Kapitels.

59 Obwohl auch mit JavaScript sowohl der Zugriff auf ein Java-API als auch der umgekehrte Weg unterstützt werden. Aber sowohl eingeschränkt als auch nicht direkt.

Das betrifft einmal den Bezug zu der Erzeugung von Objekten als auch den Umgang mit der OOP als Gesamtes. Man kann und wird beispielsweise in JavaFX mit Objekten umgehen, aber auch prozedurale Techniken verwenden. JavaFX Script ist also explizit als hybride Sprache konzipiert worden.

Und auch die genaue Syntax von JavaFX Script differiert von der von Java in vielen wichtigen Details. In mehreren Situationen fühlt man sich bei JavaFX Script eher an JavaScript denn an Java erinnert, was konsequente Java-Fans teilweise vor echte Glaubensfragen stellen wird. Andere Bestandteile von JavaFX Script sind an Notationen in XML beziehungsweise die XML-Lokalisierungssprache XPath oder auch SQL angelehnt. Und in der Behandlung von Zeichenketten[60] scheint sich Sun beispielsweise bei PHP umgesehen zu haben.

Ebenso sollten Sie beachten, dass Anweisungen in JavaFX – typisch für ein Skript – einfach sequenziell von oben nach unten abgearbeitet werden. Aber das ist für JavaFX Script nur die halbe Wahrheit. Es bedeutet nämlich nicht, dass beispielsweise oben im Quellcode notierte Aufrufe von Funktionen oder Operationen zwingend auch vorher deklariert sein müssen[61]. Der Interpreter von JavaFX sortiert gegebenenfalls den Quellcode vor der Interpretation in die richtige Reihenfolge. Er betrachtet ihn also als Ganzes.

Wenn Sie sich diese Ausführungen ansehen, haben also sowohl Einsteiger in die Welt von Java als auch erfahrene Java-Programmierer ganz spezifische (wenngleich unterschiedliche) Gründe, warum sie dieses Kapitel durcharbeiten sollten.

> > > HINWEIS

Der konkrete Blick auf den objektorientierten Ansatz von JavaFX werden wir im nächsten Kapitel in den Fokus stellen. Und JavaFX Script macht für die Erstellung von typischen Applikationen intensiven Gebrauch von GUI-Komponenten aus Swing sowie Java2D und erlaubt damit vor allen Dingen eine leichte Erstellung von grafischen Benutzerschnittstellen. Das haben wir in den ersten Beispielen ja bereits gesehen.

Aber auch dieses Thema werden wir erst in späteren Kapiteln[62] verfolgen, denn diese Technik basiert einmal explizit auf der OOP. Zum anderen blähen die notwendigen Anweisungen den Quelltext auch ziemlich auf und verstellen damit den Blick auf das, was in diesem Kapitel im Fokus stehen soll.

In dem aktuellen Kapitel behandeln wir das grundsätzliche Konzept der Technologie, die Basistypen, Variablen, Funktionen, Arrays, Ausdrücke, Operationen, Literale und andere wesentliche Aspekte der Syntax. Und die Beispiele werden entsprechend optisch nur sehr einfach sein, um nicht den Blick auf die Wesentlichkeiten zu verstellen.

60 Insbesondere der Auswertung von Ausdrücken in Zeichenketten.

61 Was bei einigen Skriptsprachen und auch älteren Programmiersprachen notwendig ist.

62 Die Erstellung von Swing-Applikationen sehen Sie in Kapitel 5 und die Verwendung von Java2D in Kapitel 8.

3.1 Das grundsätzliche Konzept von JavaFX

Wir wollen uns zuerst dem grundsätzlichen Konzept von JavaFX nähern und dabei immer wieder den Bezug zu Java selbst suchen. Sowohl deshalb, weil garantiert viele Java-Programmierer den Umstieg wagen beziehungsweise ihre Kenntnisse in der Java-Welt erweitern wollen, aber vor allem deshalb, weil sie letztendlich auch mit JavaFX echte Java-Applikationen erstellen (wenngleich mit einer übergeordneten Syntaxschicht). Java und JavaFX haben also viele Gemeinsamkeiten, aber es gibt wie schon erwähnt auch Trennendes.

Java ist eine streng objektorientierte Programmiersprache, die teils interpretiert und teils kompiliert ist. JavaFX Script hingegen ist eine sequenziell abgearbeitete Skriptsprache, die zudem nicht streng objektorientiert arbeitet. Und auch bezüglich des Umgangs mit der Interpretation muss man sich Gedanken machen.

> > > HINWEIS

In dieser Einleitung stecken nun bereits diverse Fachbegriffe, die wir uns erst einmal verdeutlichen müssen. Zwar werden erfahrene Programmierer damit klarkommen, aber diese vertieften Grundlagen in der Programmierung sollen ja im Buch explizit nicht vorausgesetzt werden.

3.1.1 Kompilierung versus Interpretation

Stellen wir zuerst das Prinzip der **Kompilierung** und der **Interpretation** gegenüber. Was hat es damit auf sich? Beide Prinzipen beschreiben Verfahren zur Übersetzung von lesbarem Klartext in Anweisungen, die ein Computer versteht.

Grundsätzlich schreibt man in so genannten höheren Programmiersprachen **Quellcode**. Das ist dieser lesbare Klartext[63]. Dieser ist mit zentralen Schlüsselwörtern, die in der Regel der englischen Sprache entnommen sind, und mit anderen Token durchsetzt. Wir haben solchen Quellcode im letzten Kapitel ja bereits mehrfach erstellt.

> > > HINWEIS

*Ein **Token** ist ein Zeichen oder eine Zeichenkombination im Quellcode. Ein Token hat eine feste Bedeutung im Rahmen einer Programmiersprache. Das kann ein Schlüsselwort wie* `for`, `do` *oder* `if` *sein, aber auch eine Syntaxstruktur wie das Plus- oder Minuszeichen, das Gleichheitszeichen, ein Klammernpaar etc.*

63 In der Regel auf Basis von ASCII-Code oder Unicode.

Dieser für Menschen[64] lesbare Quellcode muss zur Ausführung auf dem Rechner in eine Form gebracht werden, die für den Computer verständlich ist. Er muss übersetzt werden – letztendlich irgendwann einmal in etwas, was man als **Maschinencode** bezeichnet.

> > > HINWEIS

***Maschinencode** oder **Maschinensprache** ist das, was der Prozessor des Computers versteht und ausführen kann. Es handelt sich um einen Binärcode.*

Bei der Art der Übersetzung von Quellcode in Maschinencode gibt es zwei verschiedene Konzepte:

- Kompilierung
- Interpretation

Ein **Parser** verarbeitet den Quellcode vor dem konkreten Kompilieren beziehungsweise Interpretieren und zerlegt ihn anhand von Token in kleinere Einheiten. Diese werden dann vom Interpreter oder Compiler übersetzt beziehungsweise verarbeitet.

Bei der **Kompilierung** wird der gesamte Quellcode eines Programms genommen und als Ganzes übersetzt. Das macht man mithilfe eines Compilers. Das Resultat ist eine ausführbare Datei[65] oder eine binäre Bibliothek wie eine DLL, die in einer spezifischen Umgebung zum Laufen gebracht oder von einem Programm verwendet werden kann.

Bei der **Interpretation** wird der Quellcode erst zu dem Zeitpunkt übersetzt, zu dem er ausgeführt werden soll. Und dies zeilenweise von oben nach unten. Das bedeutet im allgemeinen Fall, wenn eine Zeile, die oben im Quellcode auftaucht, vom Computer abgearbeitet wird, sind Quellcodezeilen, die weiter unten im Quellcode stehen, noch nicht übersetzt. Wobei Letzteres – wie schon erwähnt – im Fall von JavaFX Script nicht ganz stimmt, denn der Interpreter[66] von JavaFX ist »intelligent« und sucht bei Bedarf auch weiter unten im Quellcode notierte Strukturen und bringt den Quellcode im Hintergrund in die richtige Reihenfolge.

Nun haben sowohl Interpretation als auch Kompilierung spezifische Vor- und Nachteile.

Kompilierter Code ist auf jeden Fall performanter und es kann gewährleistet werden, dass keine syntaktischen Fehler im resultierenden Programm vorliegen. Sollte solch ein syntaktischer Fehler im Quellcode vorliegen, wird das der Compiler entdecken und das Programm nicht zu Ende übersetzen.

64 Zumindest solchen Menschen, die ein bisschen was von Programmierung verstehen. Für andere Personen ist Quellcode oft unlesbares Zeug, das höchstens Panik auslöst. Wobei solche Programmierlaien oft auch Programmierer nicht für Menschen halten ;-).

65 Zum Beispiel eine *.exe*-Datei unter Windows.

66 Oder eigentlich der Parser, aber diese Feinheiten sollen keine Rolle spielen.

> > > HINWEIS

Fast alle Applikationen, die Sie auf Ihrem PC vorfinden, sind bereits kompiliert.

Interpretierter Code ist, da noch nicht in konkreten Maschinencode für einen spezifischen Prozessor übersetzt, universeller und leichter zu verteilen. Zudem ist die Datei, die ein Anwender erhalten muss, um ein Vielfaches kleiner als die bereits voll übersetzte, ausführbare Datei[67]. Und Interpretercode ist oft einfacher strukturiert und damit leichter zu lernen. Der Anwender, bei dem ein zu interpretierendes Programm oder Skript ausgeführt werden soll, benötigt allerdings immer einen passenden spezifischen Interpreter für eine Programmier- oder Skriptsprache.

> > > HINWEIS

Interpretation findet man sehr oft im Internet bei Skriptsprachen. So ist beispielsweise JavaScript eine interpretierte Sprache.

3.1.2 Bytecode – das Beste aus Interpretation und Kompilierung

Mittlerweile findet man sehr oft auch Mischverfahren aus Kompilierung und Interpretation. Beispielsweise auch bei Sprachen des .NET-Konzepts von Microsoft und Java, wie bereits oben erwähnt. Bei Java wird der Quellcode zuerst in so genannten **Bytecode** kompiliert, der dann in Form von `.class`-Dateien gespeichert wird und das Beste aus zwei Welten vereint. Bei diesem Bytecode handelt es sich um ein architekturneutrales Object-Code-Format. Ferner ist er binär, aber so lange nicht vollständig und lauffähig, bis er von der Java-Laufzeitumgebung interpretiert wird. Da jede Java-Laufzeitumgebung plattformspezifisch ist, arbeitet das endgültige ausgeführte Programm auf dieser spezifischen Plattform. Diese Mischform ist das Geheimnis der Plattformneutralität von Java und letztendlich auch JavaFX.

3.1.3 Die zusätzliche Architekturschicht in JavaFX

Allerdings ist bei JavaFX sozusagen noch eine weitere Ebene im Architekturmodell eingezogen. Der Quellcode in einer JavaFX Script-Applikation beschreibt keinen nativen Java-Quellcode, sondern ist ein Skriptcode mit der Dateierweiterung *.fx*, der bei der Interpretation vorhandenen Bytecode nutzen kann[68] und der zudem vor der tatsächlichen Interpretation im Hintergrund syntaktisch aufbereitet wird. Und er wird selbst – trotz der Abweichung in der Syntax des JavaFX-Quellcodes – letztendlich ebenfalls im Hintergrund wie eine gewöhnliche Java-Applikation vom JavaFX-Laufzeitsystem in Java-Bytecode übersetzt und in Form von temporären *.class*-Dateien

67 Das spielt vor allen Dingen im Internet eine große Rolle.
68 Das gesamte JavaFX-API besteht im Kern aus Java-Klassen mit der Erweiterung *.class*.

bereitgestellt. Ebenso können und werden über das JavaFX-Standard-API hinaus *.class*-Dateien auch direkt von JavaFX-Applikationen verwendet und mit ausgeliefert. Diese werden allerdings – wie auch das JavaFX-Standard-API – werden meist in Form einer komprimierten *.jar*-Datei zur Verfügung gestellt.

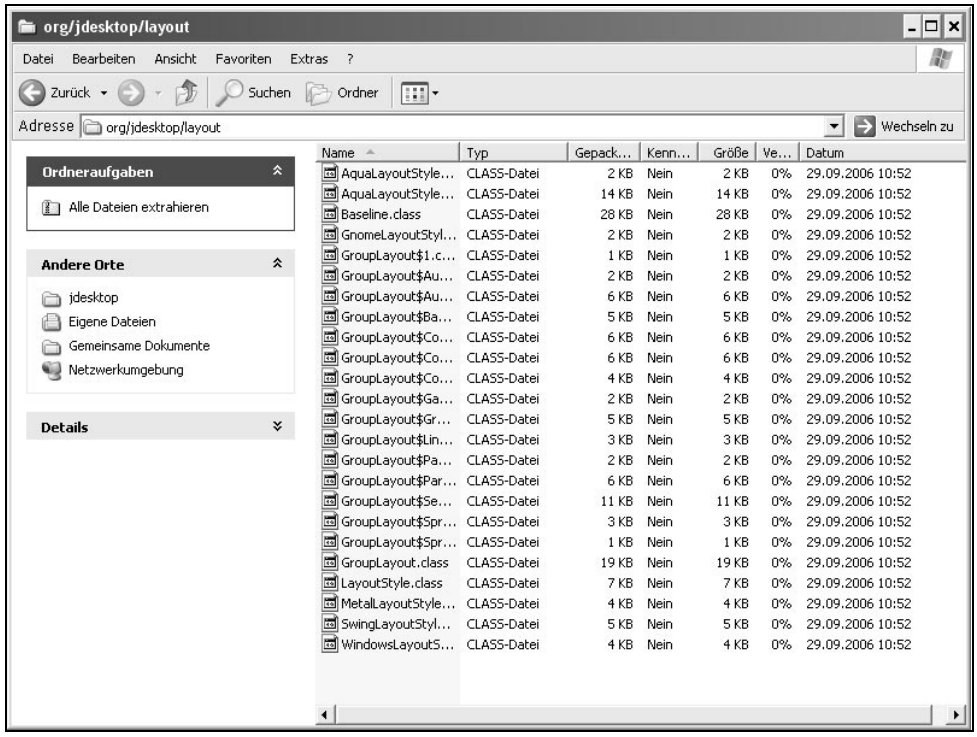

Abbildung 3.1: Eine JavaFX-Applikation liefert meist auch direkt eine Bibliothek mit gepackten .class-Dateien aus

Aber wie gesagt – bei JavaFX werden dennoch die Quellcode-Anweisungen zur Laufzeit übersetzt und das bedeutet, die Interpretation wird hier an zwei Stellen verwendet:

1. Die normale Interpretation von Java-Bytecode, wie sie auch bei allen anderen Typen an Java-Applikationen verwendet wird.
2. Die vorangehende Interpretation der Skriptbefehle von JavaFX Script, um die eigentliche Applikation zusammenzufügen.

Die Interpretation des Bytecodes bei Java-Applikationen ist mittlerweile sehr ausgereift und durch diverse Maßnahmen performant. Aber die vorangehende zusätzliche Interpretation der Skriptbefehle im Rahmen von JavaFX Script drängt unweigerlich die

Frage auf, wie es mit der Performance bei JavaFX Script aussieht, die ja allgemein bei einer Skripttechnologie ein Problem darstellt?

Nach Aussage von Sun ist jedoch die Architektur von JavaFX Script hochperformant konzipiert worden, denn alle GUI-Komponenten sowie die Objekte der Back-end-Applikation sind in purem Java implementiert und JavaFX-Skriptcode wird nur zur Erstellung, Konfiguration und Verbindung von solchen Komponenten verwendet. Diese Schritte stellen in der Gewichtung der letztendlich vom Prozessor ausgeführten Schritte einer JavaFX-Applikation also einen verschwindend geringen Anteil dar, der die Performance der gesamten Applikation nur absolut unwesentlich beeinflusst.

3.2 Ein paar elementare Grundregeln zur Syntax von JavaFX

Die Eingabe von Quellcode ist in der Programmierung in der Regel absolut nicht fehlertolerant. Das bedeutet, ein kleiner Fehler wie ein vergessenes Komma oder eine fehlende Klammer kann bereits die Übersetzung beziehungsweise Ausführung der gesamten Applikation verhindern. Das unterscheidet Skript- und Programmiersprachen beispielsweise von einer Dokumentenbeschreibungssprache wie HTML, die von einem Browser nach dem Prinzip der Fehlertoleranz interpretiert wird und durchaus massive strukturelle Fehler in einem Dokument enthalten kann. Zwar gibt es einige Programmier- und Skriptsprachen, die dennoch für einige Situationen eine gewisse Fehlertoleranz aufweisen (beispielsweise kann in einigen Programmiersprachen für eine Anweisung ein im Grunde verbindliches Semikolon zum Beenden einer Anweisung weggelassen werden, wenn es sich um die letzte Anweisung in einem Block handelt), aber diese Fälle sind streng limitiert und lassen in der Regel auch nur definierte Abweichung von einer Standardvorgabe zu[69].

Java ist nun auch syntaktisch eine sehr strenge Sprache. Das stört zwar Einsteiger oft. Erfahrenen Programmierern gefällt es jedoch sehr gut, dass es keine Ausnahmen und unklaren Situationen gibt und Java dadurch extrem logisch ist. In JavaFX Script gelten nun im Wesentlichen die gleichen Grundregeln zur Syntax wie in Java, obwohl die Strenge von Java in JavaFX Script an einigen Stellen etwas aufgeweicht und auch die Syntax teilweise etwas abgewandelt[70] wurde.

Diese Grundregeln von Java respektive JavaFX sollten Sie gleich von Anfang an beachten, um nicht unnötige Fehler zu machen, die Sie dann sowieso vor einer Übersetzung beseitigen müssen:

69 So gesehen handelt es sich also auch dann nicht um echte Fehler.
70 Hier müssen besonders erfahrene Java-Programmierer aufpassen.

- **Groß- und Kleinschreibung** im Quelltext ist grundsätzlich relevant.
- **Schlüsselwörter** der Sprache dürfen nicht als **Bezeichner** verwendet und nicht getrennt werden. Zumindest nicht ohne eine Kennzeichnung. JavaFX stellt ein Konstrukt mit so genannten **französischen Anführungszeichen** bereit, bei denen Schlüsselwörter dann doch als Bezeichner fungieren können. Aber sie sind dann speziell markiert. Darauf gehen wir natürlich detailliert ein (siehe Seite 149).
- **Zeichenketten** (in Hochkommata eingeschlossene Zeichen) dürfen im Allgemeinen in einer Programmier- oder Skriptsprache ohne spezielle Vorkehrungen nicht auf mehrere Zeilen verteilt werden. Sie können allerdings in nahezu jeder Programmier- und Skriptsprache eine Zeichenkette aus mehreren Teilen zusammensetzen. Nun geht JavaFX bei Zeichenketten einen etwas anderen Weg als fast alle anderen Programmier- und Skriptsprachen und insbesondere auch Java. JavaFX verwendet zwei Formen an Zeichenketten (mit einfachen und doppelten Hochkommata), was unterschiedliche Möglichkeiten eröffnet. Und im Gegensatz zu Java und allen mir sonst noch bekannten Programmier- und Skriptsprachen können (und werden) in JavaFX Zeichenketten Zeilenumbrüche enthalten, wenn sie in doppelte Hochkommata eingeschlossen sind[71] (siehe Seite 115)!
- Zu jeder geöffneten **Klammer** muss es genau eine zugehörige geschlossene Klammer gleichen Typs geben. Fehlende oder überflüssige Klammern sind einer der häufigsten Fehler in der Quelltexteingabe.
- **Blöcke** (Zusammenfassungen von Anweisungen) werden mit geschweiften Klammern gebildet.
- **Kommentare** werden entweder mit `//` (einzeiliger Kommentar bis zum Zeilenende) eingeleitet und mit `/*` begonnen und mit `*/` beendet (mehrzeiliger Kommentar).
- **Leerzeichen** und andere **Whitespace-Zeichen** (unsichtbare Zeichen) wie der Tabulator oder ein Zeilenumbruch können immer da in beliebiger Anzahl stehen, wo verschiedene Token getrennt werden.
- Auf Leerzeichen und andere Whitespace-Zeichen kann zwischen zu trennenden Token verzichtet werden, wenn dort ein anderes **Trennzeichen** wie ein Komma oder eine Klammer steht.
- In Java muss jede **Anweisung** mit einem Semikolon beendet werden. Das ist in JavaFX auch der Fall, obwohl es manchmal nicht so scheint. Das Problem resultiert daraus, dass Anweisungen in JavaFX eine etwas andere Struktur als in Java haben können und es damit für Umsteiger von anderen Sprachen etwas verschleiert wird, wo genau eine Anweisung zu Ende ist.

71 Eine Zeichenkette wird also im Editor auf mehrere Zeilen verteilt.

3.3 Die Basistypen von JavaFX Script und Variablen

Ein **Typ** beziehungsweise **Datentyp** gibt in einer Computersprache an, wie ein einfaches Objekt (wie zum Beispiel eine **Variable** oder **Literal**) im Speicher des Computers dargestellt wird.

> > > HINWEIS

Ich stelle Programmiereinsteiger in meinen Programmierkursen fast immer die Frage, was eine Variable ist. In der Regel erhalte ich als Antwort eine Definition der Art, dass eine Variable irgendetwas Veränderbares ist. Aus der Schulmathematik erinnern sich fast alle Teilnehmer an den Begriff einer Variablen.

*In der Programmierung muss man jedoch etwas anders an die Definition herangehen. Eine **Variable** sollte man als eine Stelle im Hauptspeicher des Rechners verstehen, deren Inhalt man auslesen und durch Zugriff aus einer Programmiersprache verändern kann.*

*Ein **Literal** ist hingegen ein Wert wie eine Zahl, ein einzelnes Zeichen oder eine Zeichenkette. Literale werden in Ausdrücken verwendet und oft Variablen zugewiesen.*

Beides – Variablen beziehungsweise deren Bezeichner und Literale – sind Token, die in jeder Programmier- und Skriptsprache vorkommen.

3.3.1 Der Datentyp

Ein **Datentyp** legt in einer Programmiersprache fest, wie viel Platz für ein einfaches Objekt im Speicher des Computers reserviert wird und welche Arten von Werten in diesem Speicherbereich abgelegt werden können. Dies impliziert für numerische Werte auch einen **Wertebereich**. Der Datentyp enthält ebenfalls Informationen darüber, welche Operationen mit und an ihm (beziehungsweise dem einfachen Objekt) ausgeführt werden können. Viele Computersprachen lassen es beispielsweise nicht zu, dass mit einer alphanumerischen Zeichenfolge (also Texten) direkte arithmetische Operationen durchgeführt werden[72].

Java selbst besitzt nun acht **primitive Datentypen**[73], die explizit plattformunabhängig sind, und drei so genannte **Referenzdatentypen**.

72 Das ist meist auch sinnvoll. Oder was soll `"Apfel" geteilt durch "Birne"` ergeben ;-)?

73 Im Detail sind das in Java die ganzzahligen numerischen Typen `byte`, `short`, `int` und `long` mit Vorzeichen, die beiden Gleitkommatypen `float` und `double`, `char` für Zeichen über eine ganzzahlige 16 Bit-Unicodedarstellung (ohne Vorzeichen) und der 1 Bit große Wahrheitswert `boolean`.

> > > HINWEIS

*Ein **primitiver Datentyp** (auch einfacher Datentyp genannt) ist direkt in die Programmiersprache eingebaut. Etwa die Darstellung für eine Zahl, einen Wahrheitswert oder einen Buchstaben. Streng genommen darf es in objektorientierten Programmiersprachen gar keine primitiven Datentypen geben, aber Sun hat ob der besseren Verwendbarkeit und einem sinnvollen Bezug zur traditionellen Programmierung bei der Einführung von Java mit diesem Dogma gebrochen.*

*Ein **Referenzdatentyp** ist hingegen ein **Verweis** (oft auch **Zeiger** oder **Pointer** genannt) auf einen bestimmten Speicherbereich. Damit werden in der Regel Objekte referenziert. In Java sind alle Referenztypen 32 Bit groß.*

Ihnen sollte auffallen, dass also die oft zu hörende Aussage »Java hat keine Pointer« so nicht richtig ist. Aber Java verwendet die Pointer über diese Referenztypen indirekt und versperrt einem Programmierer den direkten Zugriff auf Speicherbereiche. Das ist ein immenser Stabilitäts- und Sicherheitsvorsprung von Sprachen, bei denen ein Programmierer über Pointer direkt auf Speicherbereichen agiert.

Im Gegensatz zu Java unterstützt JavaFX nur vier primitive Basistypen, deren Gegenstück zudem in Java teils über Referenztypen und nicht primitive Datentypen abgebildet werden. Und wenn man genauer hinschaut, verwaltet JavaFX diese vier primitiven Basistypen als Referenztypen[74], was JavaFX damit massiv von Java unterscheidet:

- Der Datentyp `String` dient zur Beschreibung von **Zeichenketten**. Das sind in JavaFX alle Texte, die in **doppelte oder einfache Hochkommata** eingeschlossen werden. An dieser Stelle sehen Leser mit Java-Erfahrung einen deutlichen Unterschied zu Java, wo Zeichenketten **ausschließlich** in doppelte Hochkommata eingeschlossen werden dürfen und einfache Hochkommata ausschließlich für Character (einzelne Zeichen – in Java ist das der primitive Datentyp `char`) verwendet werden! Aber beispielsweise in JavaScript oder PHP kennt man ebenso diese zweifache Verwendung der Kennzeichnung von Zeichenketten. Da JavaFX jedoch auch einzelne Zeichen[75] grundsätzlich als Zeichenketten betrachtet, ist hier eine Unterscheidung zwischen einfachen und doppelten Hochkommata nicht notwendig. Und wie auch in PHP werden in JavaFX Zeichenketten in doppelten und einfachen Hochkommata vom System etwas anders behandelt. Die doppelte Form der Kennzeichnung von Zeichenketten in JavaFX eröffnet einige interessante Möglichkeiten[76].

74 Das bedeutet als Konsequenz, dass sie Methoden bereitstellen.

75 Diese werden in Java wie gesagt über den primitiven Datentyp `char` abgebildet und streng von Zeichenketten (das sind in Java im Gegensatz dazu keine primitiven Datentypen, sondern Objekte) unterschieden.

76 Darauf kommen wir noch genauer zu sprechen (siehe Seite 113).

- Der Datentyp `Boolean` ist ein Typ für einen **Wahrheitswert**. Das sind die Token `true` für wahr und `false` für falsch. Solche Wahrheitswerte werden im Wesentlichen bei Vergleichen verwendet. Im Grunde ist hier kein wesentlicher Unterschied zu Java zu sehen, nur kennt Java neben der Wrapper-Klasse[77] `Boolean` auch den primitiven Datentyp `boolean`, der in der Praxis auch weit häufiger verwendet wird.
- Der Datentyp `Number` dient zur Beschreibung von allgemeinen numerischen Werten über eine Objektdarstellung. In Java gibt es diese Wrapper-Klasse für Zahlen auch, aber in der Praxis verwendet man meist die primitiven Datentypen sowie gelegentlich abgeleitete Klassen von `Number`.
- Der Datentyp `Integer` dient zur Beschreibung von primitiven ganzzahligen numerischen Werten mit Vorzeichen und großen Ganzzahlen. Auch hier gibt es ein direktes Gegenstück in Java. Daneben gibt es jedoch in Java zusätzlich die Wrapper-Klassen `Byte`, `Short` und `Long`, um Gegenstück für alle ganzzahligen primitiven Datentypen bereitzustellen. Und auch bei ganzen Zahlen verwendet man in der Praxis in Java fast immer die primitiven Datentypen.

Diese Basistypen von JavaFX korrespondieren mit den Datentypen von Java also wie folgt:

JavaFX	Java
`String`	`java.lang.String`
`Boolean`	`java.lang.Boolean`,
	`boolean`
`Number`	`java.lang.Number`, `java.lang.Integer`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`,
	`byte`, `short`, `int`, `long`, `float`, `double`
`Integer`	`java.lang.Integer`, `java.lang.Byte`, `java.lang.Short`, `java.lang.Long`, `java.math.BigInteger`,
	`byte`, `short`, `int`, `long`

Tabelle 3.1: Die Datentypen von JavaFX versus den Datentypen von Java

Beispiele:

```
var s = "Hallo";
var n = 1.5;
var b = true;
```

Listing 3.1: Datentypen bei Variablendeklarationen

77 Eine **Wrapper-Klasse** (kurz Wrapper) in Java ummantelt einen primitiven Datentyp und erlaubt so den Austausch von Werten zwischen primitiven Datentypen und Klassen beziehungsweise Objekten.

> > > HINWEIS

Nun wird erfahrenen Java-Programmierern auffallen, dass die Kompatibilität der Datentypen zwischen Java und JavaFX einige automatische Anpassungen notwendig macht, wenn Argumente an eine Java-Methode übergeben werden oder ein entsprechender Rückgabewert geliefert wird. Immerhin muss beispielsweise ein Referenztyp `Integer` in JavaFX in Java über einen primitiven Datentyp `int` verarbeitet werden oder umgekehrt. Das JavaFX-System löst die Umwandlung automatisch aus.

> > > HINWEIS

*Eine **Methode** können Sie sich erst einmal einfach nur als ein Unterprogramm vorstellen.*

3.3.2 Die Variablendeklaration in JavaFX

Die skizzierten Beispiele oben haben bereits angedeutet, dass in JavaFX ein Schlüsselwort `var` im Einsatz ist, das in dieser Form im reinen Java nicht vorkommt[78]. Allerdings sollten JavaScript-Programmierer dieses Schlüsselwort und den Einsatz kennen – das Verfahren in JavaFX Script ist verwandt. Aber glücklicherweise ist der Einsatz des Schlüsselwortes `var` in JavaFX strenger reglementiert als in JavaScript. So muss es für einen bestimmten Anweisungstyp zwingend notiert werden und ist nicht optional wie in JavaScript.

Mit diesem Schlüsselwort `var` führen Sie eine neue Variable ein. Eine solche Anweisung nennt man eine **Deklaration** der Variablen.

Beispiel:

```
var z;
```

Listing 3.2: Eine Variablendeklaration ohne Festlegung des Datentyps

Bei der Deklaration können Sie auch gleich den Datentyp der Variablen spezifizieren.

Beispiel:

```
var z: String;
```

Listing 3.3: Eine Variablendeklaration mit Festlegung des Datentyps

Wie Sie sehen, folgt dem Schlüsselwort `var` der **Bezeichner**[79] der Variablen, dann ein Doppelpunkt und anschließend der Datentyp.

78 Obwohl es in Java reserviert ist.

79 Also der Name, unter dem sie in den folgenden Anweisungen des Quelltexts angesprochen werden soll.

Deklaration mit Initialisierung

Nun können Sie auch bei der Deklaration einer Variablen bereits einen Wert geben. Das nennt man die **Initialisierung**.

Beispiel:

```
var z: String = "RJS"
```

Listing 3.4: Eine Variablendeklaration mit Festlegung des Datentyps in Initialisierung

Mit dem Gleichheitszeichen wird der Variablen ein Anfangswert zugewiesen. Der Anfangswert kann ein Literal sein, aber auch berechnet oder als Ergebnis einer Funktion geliefert werden.

Und da auch keine Wertzuweisung am Anfang erfolgen muss, könnte man das auch auf zwei Zeilen verteilen:

```
var nums; // Deklaration
nums  = 1; // Initialisierung
```

Listing 3.5: Ein Beispiel für Verteilung der Deklaration und der Wertzuweisung

! ! ! ACHTUNG

Diese Form der Variablendeklaration unterscheidet sich massiv von der Syntax in Java.

Deklaration ohne explizite Typfestlegung

Die explizite Festlegung des Datentyps bei der Deklaration ist nun optional in JavaFX. Es gibt einige Abwandlungen von dieser Form.

Wenn Sie den Datentyp bei der Deklaration nicht auf diese Weise (also einem Doppelpunkt nachgestellt) festlegen, wird der JavaFX-Interpreter den Datentyp der Variablen aus dem Datentyp des zugewiesenen Literals oder Ausdrucks **implizit** festlegen.

Da man die Wertzuweisung einer Variablen auch nach einer Deklaration vornehmen kann, bedeutet das, dass die Variable bis zum Zeitpunkt der impliziten Typzuweisung durch den Typ eines Literals oder Ausdrucks keinen sinnvollen Wert repräsentiert. Allerdings sollten Sie sich noch einmal vergegenwärtigen, dass JavaFX ausschließlich mit Referenztypen arbeitet, und die sind im Typ und der Größe eindeutig. Die Referenzvariable weist also in so einem Fall einfach noch nicht gezielt auf einen bestimmten Speicherbereich. Sie ist aber in der Grundstruktur festgelegt.

> > > HINWEIS

Man kann durchaus geteilter Meinung sein, ob die optionale Verwendung einer expliziten Festlegung eines Datentyps in einer Programmierer- oder Skriptsprache nun von Vorteil oder von Nachteil ist. Auf der einen Seite erleichtert eine (optionale) implizite Festlegung des Datentyps für Einsteiger den Umgang mit Variablen erheblich. Auch wird der Quellcode kleiner. Und zudem kann man dynamisch auf gewisse Situationen reagieren, bei denen sich ein Datentyp einer Variablen erst zur Laufzeit ergibt.

Auf der anderen Seite wird die Klarheit und Wartbarkeit eines Programms beziehungsweise Skripts reduziert. Gerade Programmierer aus strengen Sprachen wie Java schätzen den unabdingbaren Zwang, bei der Deklaration den Datentyp angeben zu müssen[80]. Je weniger Ausnahmen und verschieden zu behandelnde Situation es in einer Sprache gibt, desto logischer ist die Sprache.

In JavaFX kontrolliert das System allerdings bei einer Angabe des Datentyps für eine Variable, ob der Datentyp des zugewiesenen Werts mit diesem übereinstimmt. Ebenso kann in JavaFX ein einmal zugewiesener Datentyp für eine Variable nicht mehr im Laufe des Skripts geändert werden! Dies ist in so genannten lose typisierten Sprachen wie JavaScript oder PHP beispielsweise möglich. Dort kann man auch eine Variable ohne konkrete Deklaration über eine Wertzuweisung zu einem bisher noch nicht verwendeten Bezeichner einführen[81]. Das ist ebenso in JavaFX nicht möglich, was auch für erfahrene Java-Programmierer die Bauchschmerzen in Grenzen halten sollte.

! ! ! ACHTUNG

Eine Deklaration von mehreren Variablen in einer Anweisung (was in Java möglich ist) geht in JavaFX nicht. Weder mit noch ohne direkte Initialisierung. Also etwas der folgenden Art wäre in JavaFX falsch:

```
var a, v;
var a = 5, b = 6;
```

Listing 3.6: Das geht in JavaFX nicht

3.3.3 Festlegung der Kardinalität einer Variable

In JavaFX ist es nun möglich, bei der Deklaration einer Variablen die so genannte **Kardinalität** (Häufigkeit des Vorkommens) der Variable festzulegen.

80 Dazu zähle ich mich persönlich auch, obwohl ich sehr viel mit Sprachen wie JavaScript und PHP arbeite, in denen so eine implizite Festlegung des Datentyps sogar nur ausschließlich möglich ist.

81 Dieses Verhalten erleichtert Einsteigern den Umgang mit Variablen noch einmal ziemlich. Auf der anderen Seite wird die Wartbarkeit jedoch genauso extrem erschwert (es können unbeabsichtigt Variablen entstehen, wo man einer bestehenden Variable eigentlich nur einen neuen Wert zuweisen wollte). Und im Fehlerfall kann die Suche nach einem Fehler äußerst diffizil werden.

> > > HINWEIS

*So etwas in der Art ist beispielsweise in der **DTD** (**Document Type Definition**) vollkommen analog möglich. Damit kann zum Beispiel für ein XML-Dokument die Kardinalität eines Elements festgelegt werden. In Programmier- und Skriptsprachen ist so etwas jedoch eher ungewöhnlich. Die Kardinalität wird in JavaFX vor allem bei der Beschreibung von Attributen einer Klasse eine wichtige Rolle spielen.*

Um die Kardinalität bei der Deklaration festzulegen, verwenden Sie einen der Operatoren `?`, `+`, oder `*`, wie sie genauso beispielsweise in einer DTD (allerdings mit kleinen Abweichungen in der Bedeutung) verwendet werden.

> > > HINWEIS

*Ein **Operator** ist ein Token, über den eine bestimmte Operation angezeigt wird, die ein Interpreter oder Compiler ausführen soll. Etwa eine Wertzuweisung, eine mathematische Operation, ein Vergleich etc.*

Operator	Bedeutung
`?`	Optional – kann also auch fehlen.
`+`	Mindestens einmal oder mehr
`*`	Null oder mehrmals

Tabelle 3.2: Die Operatoren zur Angabe der Kardinalität einer Variablen

Beispiel:

```
var zahlen:Number* = 1;
```

Listing 3.7: Ein Beispiel für die Angabe einer Kardinalität

In dem Listing wird einen Variable `zahlen` eingeführt und initialisiert, deren Werte vom Typ `Number` und null oder mehrfach vorkommen.

Da in JavaFX neben der Kardinalität auch die Typangabe optional ist, könnte man für eine Variablendeklaration mit Initialisierung auch Folgendes schreiben:

```
var zahlen* = 1;
```

Listing 3.8: Ein Beispiel für die indirekte Angabe einer Kardinalität

3.4 Funktionen und Prozeduren

Erfahrene und vor allem überzeugte Java-Programmierer (oder allgemein Programmierer aus dem Umfeld der streng objektorientierten Programmierung) werden bei den Schlagwörtern **Funktionen** und **Prozeduren** im Rahmen des Java-Universums wahrscheinlich erst einmal eine schwere Sinnkrise bekommen. Es zählt ja zu den elementarsten Paradigmen der objektorientierten Programmierung, dass es dort keinerlei objektungebundenen Unterprogramme gibt und geben darf. Manche OO-Prediger gehen sogar noch weiter und sagen, dass der große Vorteil der OOP neben dem Weglassen von globalen Variablen der Verzicht auf Funktionen und Prozeduren ist! Denn damit wird die Wiederverwendbarkeit von Programmbestandteilen gewährleistet.

> > > HINWEIS

*Eine **Funktion** stellt in der klassischen (oder auch prozedural genannten) Programmierung ein Unterprogramm dar. Funktionen bildeten den Kern der so genannten **modularen Programmierung**, die in der »Frühzeit« der EDV eine erste Strukturierung von größeren Programmen in einzelne Module mit einer begrenzten Wiederverwertbarkeit gestattete. Im Gegensatz zu einer Methode in der OOP ist eine Funktion jedoch nicht an ein Objekt gebunden, sondern kommt als eigenständige Struktur daher, die direkt über ihren Bezeichner aufgerufen werden kann – ohne einem Objekt dazu eine Botschaft zu schicken (d. h. ohne ein vorangestelltes Objekt). Eine Funktion liefert nach Beendigung ihrer Tätigkeit einen so genannten **Rückgabewert** (ein Ergebnis) an den Aufrufer der Funktion. In JavaFX **muss** jede Funktion einen Rückgabewert liefern.*

> > > HINWEIS

*Eine **Prozedur** ist im Allgemeinen eine Funktion ohne Rückgabewert. In JavaFX ist das Verständnis einer Prozedur aber etwas anders. Hier **kann** eine Prozedur durchaus einen Rückgabewert liefern, **muss** es aber **nicht**. Diese Verwendung des Begriffs einer Prozedur unterscheidet sich von den meisten anderen prozeduralen Sprachen. Da man in JavaFX bei einer Prozedur auch von einer **Operation** spricht, ist die konsequente Verwendung dieses Begriffs meines Erachtens sinnvoller.*

Aber um es noch einmal zu betonen – JavaFX Script ist nicht streng objektorientiert, sondern eben eine Skriptsprache, die auf einer strengen OO-Welt operiert und ein eigenes Sprachkonzept darauf aufsetzt. Und dieses arbeitet nun mal mit Funktionen und Prozeduren[82].

82 Die Experten bei Sun werden sich schon was dabei gedacht haben. Und auch strukturell mit JavaFX Script verwandte Skriptsprachen wie JavaScript oder PHP arbeiten ebenfalls mit Funktionen als auch Objekten.

3.4.1 Funktionen in JavaFX

JavaFX-Funktionen repräsentieren eine reine funktionale Teilmenge der Programmiersprache selbst. Der Körper einer Funktion darf nur eine Reihe an Variablendeklaration und eine `return`-Anweisung enthalten. Dies gestattet natürlich nur eine sehr eingeschränkte Verwendung von Funktionen.

> > > HINWEIS

*Eine `return`-Anweisung in einer Funktion oder Methode liefert ein Ergebnis (den **Rückgabewert**) an den Aufrufer der Funktion respektive Methode, der dann verarbeitet werden kann. Entweder erfolgt die Verarbeitung direkt oder der Rückgabewert wird einer Variablen zugewiesen. Es handelt sich hierbei um eine so genannte **Sprunganweisung**. Diese gibt in einem Programm oder Skript den Programmfluss bei ihrem Aufruf unmittelbar entweder an den Anfang oder das Ende des derzeitigen Blocks weiter (siehe auch Seite 125).*

Die Deklaration einer Funktion erfolgt mit dem Schlüsselwort `function`, dem der Bezeichner der Funktion und ein Klammernpaar folgen. In den Klammern kann man optional Übergabewerte an die Funktion angeben, die dann innerhalb der Funktion als **lokale Variablen** gelten und nur dort zugänglich sind.

> > > HINWEIS

*Eine **lokale Variable** ist nur innerhalb der Struktur verfügbar, in der sie definiert wurde. In der Regel ist eine solche Struktur eine Funktion beziehungsweise allgemein Unterprogramm, aber es gibt auch Sprachen (etwa Java oder JavaFX), in denen eine lokale Variable auf einen Block oder einen begrenzten Bereich beschränkt werden kann.*

Beachten Sie, dass kein Datentyp für den Rückgabewert als auch die Parameter angegeben wird, da JavaFX eine statisch typisierte Sprache ist.

Aber schauen wir uns eine Funktionsdeklaration samt dem Aufruf in einer kleinen, aber vollständigen JavaFX-Applikation einmal an. In dem Beispiel soll mit dem Aufruf zweier Funktionen ein Ergebnis berechnet werden, das dann in der Folge in einem Label innerhalb eines kleinen Fensters angezeigt werden soll (den Quelltext finden Sie auf der Buch-CD im Ordner *javafxapp4*):

```
01 import javafx.ui.Frame;
02 import javafx.ui.Label;
03
04 function zahl(a, b) {
05   var x = a + b;
06   var y = a - b;
07   var ergeb = (sq(x) / sq (y)).toString();
08   return ergeb;
09 }
```

```
10 function sq(n) {
11   return n * n;
12 }
13 Frame {
14   content: Label {
15   text: zahl(4,5)
16   width: 100
17   }
18   visible: true
19 }
```

Listing 3.9: Eine Funktion im Einsatz

In den Zeilen 4 bis 9 wird eine Funktion `zahl()` mit Übergabewerten an die Funktion definiert. Die Parameter mit den Bezeichnern `a` und `b` stellen in der Funktion wie gesagt lokale Variablen innerhalb der Funktion dar, die in unserem Beispiel für interne Berechnungen verwendet werden. Auch wenn Sie arithmetische Operatoren im Detail noch kennen gelernt haben, werden Ihnen die Addition und Subtraktion in Zeile 5 und 6 mit Sicherheit vertraut sein. Die Ergebnisse der jeweiligen Operationen werden in den Variablen `x` und `y`, die in der Funktion lokal eingeführt werden, gespeichert.

In Zeile 7 wird eine weitere Variable mit dem Bezeichner `ergeb` eingeführt, der das Ergebnis einer Berechnung zugewiesen wird. Die Berechnung verwendet den Rückgabewert einer weiteren Funktion mit Namen `sq()`, die in den Zeilen 10 bis 12 definiert ist und das Quadrat des Übergabewertes als Rückgabewert liefert.

Beachten Sie, dass der Datentyp der Variablen `ergeb` ein String ist. Diesen Datentyp benötigen wir, wenn wir das Ergebnis der Berechnung der Eigenschaft `text` eines Labels zuweisen wollen (das sehen Sie in Zeile 15). Da numerische Typen im JavaFX als Objekte zu sehen sind, besitzen sie auch Methoden. Insbesondere besitzt jedes Objekt wie in Java selbst eine Methode `toString()`, die die Stringrepräsentation des gespeicherten Werts des Objektes liefert. Diese Methode wenden wir in dem Beispiel an und die resultierende Stringrepräsentation wird als Rückgabewert in Zeile 8 an den Aufrufer weitergegeben.

Von Zeile 13 bis Zeile 19 sehen Sie die Syntax, um in JavaFX ein Fenster mit einem Label aufzubauen, mit gewissen Eigenschaften zu versehen und anzuzeigen. Genau genommen wird hier über eine so genannte deklarative Syntax ein Objekt der Klasse `javafx.ui.Frame` erzeugt. Darauf gehen wir im nächsten Kapitel genauer ein.

Abbildung 3.2: Die Zahl wurde unter Verwendung von zwei Funktionen ermittelt

> > > HINWEIS

Die Reihenfolge, in der Funktionen im Quelltext deklariert werden, ist im Allgemeinen nicht von Bedeutung. Wie schon angedeutet, formt der Interpreter bei Bedarf den Quellcode vor der Interpretation intelligent um. Das heißt, die nachfolgende Umstrukturierung des Beispiels eben ist vollkommen äquivalent. Die aufgerufenen Funktionen werden erst hinter der Stelle deklariert, an der sie aufgerufen werden.

```
01 import javafx.ui.Frame;
02 import javafx.ui.Label;
03
04 Frame {
05   content: Label {
06   text: zahl(4,5)
07   width: 100
08   }
09   visible: true
10 }
11 function sq(n) {
12   return n * n;
13 }
14 function zahl(a,b) {
15   var x = a + b;
16   var y = a - b;
17   var ergeb = (sq(x) / sq (y)).toString();
18   return  ergeb;
19 }
```

Listing 3.10: Obwohl die Funktionen erst hinter der Stelle deklariert werden, an der sie aufgerufen werden, funktioniert die Sache

3.4.2 Prozeduren beziehungsweise Operationen

Neben Funktionen stellt JavaFX wie gesagt auch explizit **Prozeduren** zur Verfügung. Es ist bemerkenswert, dass eine moderne Skriptsprache damit scheinbar wieder Gräben aufreißt, die man vor vielen Jahren bereits mühsam zugeschüttet hatte. Die explizite Trennung zwischen einer Prozedur und einer Funktion wurde damals in den meisten Skript- als auch Programmiersprachen aufgehoben, da die Unterscheidung, ob ein Rückgabewert vorhanden ist oder nicht, für die meisten Programmierer mehr Ärger als Vorteile bedeutet hat. Und gerade in der objektorientierten Programmierung unterscheidet man bei Methoden ganz bewusst nicht mehr zwischen diesen beiden Arten eines Unterprogramms.

Aber in JavaFX wird wieder bewusst eine solche Unterscheidung vollzogen. Prozeduren, die wie schon erwähnt in JavaFX auch **Operationen**[83] genannt werden und mit dem optionalen Rückgabewert auch von klassischen Prozeduren abweichen, sind viel

83 Darauf kommen wir gleich genauer zu sprechen.

flexibler als Funktionen. Sie können im Gegensatz zu Funktionen in JavaFX eine beliebige Art an Anweisungen enthalten – inklusive Bedingungen, Schleifen, Befehle zur Ausnahmebehandlung etc.

In JavaFX werden Prozeduren oder Operationen mit einem eigenen Schlüsselwort bei der Deklaration eingeleitet[84] – `operation`.

Sehen wir uns ein Beispiel an (das Beispiel ist auf der Buch-CD im Ordner *javafxoperation1* zu finden):

```
01 import java.lang.StringIndexOutOfBoundsException;
02 import java.lang.System;
03 import java.lang.StringIndexOutOfBoundsException;
04
05 operation substring(s:String, n:Number): String {
06   try {
07     return s.substring(n);
08   }
09   catch (e:StringIndexOutOfBoundsException) {
10     return "Der Text hat weniger Zeichen als der Index";
11   }
12  }
13
14 System.out.println(substring("Hau Dich raus", 1));
15 System.out.println(substring("Hau Dich raus", 5));
16 System.out.println(substring("Hau Dich raus", 10));
17 System.out.println(substring("Hau Dich raus", 15));
```

Listing 3.11: Die Definition und Anwendung einer Prozedur in JavaFX

Das Beispiel definiert eine Prozedur mit Namen `substring()`. Die Prozedur hat zwei Argumente. Das erste Argument ist ein String, das zweite Argument eine Zahl vom Typ `Number`. Die Prozedur liefert einen Teil String des ersten Parameters ab der spezifizierten Stelle, in dem zweiten Argument angegeben wird.

> > > HINWEIS

*In dem Beispiel kommt so eine genannte **Ausnahmebehandlung** zum Einsatz. Die Grundlagen dazu haben wir bisher noch nicht erarbeitet. Sie folgen aber im Laufe des Buchs. Für jetzt können Sie den Aufbau der Prozedur so verstehen, dass nur dann ein Teilstring geliefert wird, wenn der als zweites Argument übergebene Index nicht größer als die Länge des Strings ist, der als erstes Argument der Prozedur übergeben wird.*

Beachten Sie, dass die Prozedur in Abhängigkeit von der Situation zwei verschiedene Rückgabewerte liefert (in Zeile 7 den Teil-String und in Zeile 10 eine Fehlermeldung).

84 Nicht mit `function`.

Das ist die Ausgabe des Beispiels:

au Dich raus

ich raus

aus

Der Text hat weniger Zeichen als der Index

Im Einzelnen können Sie innerhalb einer Prozedur die folgenden Anweisungen notieren[85]:

- Ausdruckanweisungen
- Entscheidungsanweisungen
- Wiederholungsanweisungen
- Anweisungen zur Ausnahmebehandlung (`try-catch`)
- Sprunganweisungen (`return`, `throw`, `break` und `continue`)
- Operatoren und Ausdrücke

3.5 Operatoren und Ausdrücke

Ein Operator ist ein Token in einer Programmiersprache, der eine bestimmte Operation auslöst. Zum Beispiel die Addition von zwei Operanden, die rechts und links von einem Operator notiert werden. Oder es werden zwei Operanden miteinander verglichen. Oder es wird mit einem Operator aus einem positiven numerischen Operanden das negative Gegenstück. Operatoren sind allgemein das wichtigste Mittel, um so genannte **Ausdrücke** formulieren zu können.

Die Operatoren in JavaFX sind nun zu einem gewissen Teil identisch mit denen in Java. Genau genommen stimmen die konkreten Zeichen für viele Operatoren mit denen in Java überein, wenn diese in beiden Sprachen identisch vorhanden sind.

Allerdings nicht in jedem Fall!

JavaFX stellt jedoch überdies einige Operatoren bereit, die es so in Java überhaupt nicht gibt. Das bedeutet aber nicht in jedem Fall, dass es diese Funktionalität in Java nicht gibt. Nur werden dort diverse Operatoren aus JavaFX in Form von Eigenschaften oder Methoden beziehungsweise anderen Sprachkonstrukten zur Verfügung gestellt.

85 Was es mit den verschiedenen Anweisungstypen auf sich hat, wird im Laufe des Kapitels im Detail geklärt.

> > > HINWEIS

Der Grund für die Übereinstimmung von Operatoren in vielen Sprachen ist die gemeinsame Sprachbasis C. Sie werden daher in sehr vielen Programmier- und Skriptsprachen (PHP, JavaScript, Perl, C, C++, C#, etc.) die gleichen Operatoren vorfinden, da sie meist auf C zurückzuführen sind. Für JavaFX muss man jedoch festhalten, dass es diverse Abweichungen von der sonst allgemein üblichen C-Syntax gibt. Und diese Abweichungen sind so gemacht, dass die entsprechenden Operatoren in Java respektive C auch gar nicht als Alternative verwendet werden können (was in anderen Sprachen oft möglich ist – hier gehen meist die C-Varianten als auch – oft sprechendere – alternative Schreibweisen).

Auch hier muss ich anmerken, dass sich mir als Java-Programmierer beim ersten Kontakt mit JavaFX die Gründe für diese Abweichungen nicht erschließen wollten. Je länger ich aber darüber nachgedacht habe, desto mehr konnte ich die Überlegung von Sun nachvollziehen[86]. JavaFX soll sich auch Leuten erschließen, die nicht permanent mit C-lastigen Sprachen programmieren. Bei der Wahl der Token für verschiedene Operatoren ist man zum Teil auf Zeichen der Mathematik ausgewichen (etwa der Operator für Ungleichheit) oder hat versucht, sprechende Token für manche Operatoren zu finden. Es ist garantiert leichter für einen Java-Programmierer, sich diese Token zu merken als für ein Programmierneuling die Token aus C.

JavaFX stellt konkret die nachfolgend besprochenen Operatoren bereit.

3.5.1 Relationale Operatoren

Relationale Operatoren (**Vergleichsoperatoren**) werden bei Vergleichen von Operanden verwendet. Dies macht man beispielsweise bei Bedingungen in Entscheidungsstrukturen oder Schleifen.

Operator in JavaFX	Bedeutung	Entsprechender Operator in Java
==	Gleichheit.	==
<>	Ungleichheit. Achtung! In fast allen vergleichbaren Sprachen wird die Variante verwendet, wie sie in Java üblich ist[87].	!=
<	Kleiner als	<
>	Größer als	>
<=	Kleiner als oder gleich	<=
>=	Größer als oder gleich	>=

Tabelle 3.3: Die relationalen Operatoren in JavaFX

86 Zumindest glaube ich das.

87 Mir fällt aber Pascal ein, wo wie in JavaFX ebenso der Token <> Verwendung für die Ungleichheit findet.

3.5.2 Logische Operatoren

Die logischen Operatoren (auch Booleschen[88] oder Boolean Operatoren genannt) dienen zur Verknüpfung von logischen Ausdrücken, wie sie beispielsweise ein Vergleich liefert. JavaFX versucht diese logischen Operatoren mit möglichst sprechenden Token umzusetzen. Dabei werden Token verwendet, die auch in vielen anderen Sprachen als Ersatz für die C-Syntax verfügbar sind. JavaFX unterstützt aber explizit diese C-Syntax als Alternative nicht und stellt ausschließlich die sprechenden Token zur Verfügung.

Operator in JavaFX	Bedeutung	Entsprechender Operator in Java
`and`	Logisches Und	`&&`
`or`	Logisches Oder	`||`
`not`	Logische Negation	`!`

Tabelle 3.4: Die logischen Operatoren in JavaFX

! ! ! ACHTUNG

Beachten Sie, dass auch bei den logischen Operatoren Groß- und Kleinschreibung von Bedeutung ist. `AND` *ist* ***kein*** *gültiger Token für einen logischen Operator.*

3.5.3 Arithmetische Operatoren

Die arithmetischen Operatoren dienen zur Berechnung. Hier finden Sie auch keinerlei Abweichungen zwischen JavaFX und Java (beziehungsweise der C-Syntax, aus der diese Operatoren eigentlich stammen).

Operator in JavaFX und Java	Bedeutung
`+`	Addition
`-`	Subtraktion und Negierung.
`*`	Multiplikation
`/`	Division
`%`	Modulo – der Restwert nach einer Division des linken Operanden durch den rechten Operanden
`+=`	Addition mit gleichzeitiger Zuweisung
`-=`	Subtraktion mit gleichzeitiger Zuweisung
`*=`	Multiplikation mit gleichzeitiger Zuweisung
`/=`	Division mit gleichzeitiger Zuweisung

Tabelle 3.5: Die arithmetischen Operatoren

88 Nach dem Mathematiker Boole

Operator in JavaFX und Java	Bedeutung
%=	Modulo mit gleichzeitiger Zuweisung
++	Inkrement
--	Dekrement

Tabelle 3.5: Die arithmetischen Operatoren (Fortsetzung)

Einige der arithmetischen Operatoren dürften aus der Schulmathematik bekannt sein. Andere wiederum nicht. Die arithmetischen Operatoren beinhalten für Einsteiger mit Programmierung sogar einige Tücken. Wir wollen deshalb einige der Operatoren in einem praktischen Beispiel besprechen. Konkret betrifft das vor allem den Modulooperator und die arithmetischen Zuweisungsoperatoren sowie die Inkrement- und Dekrementoperatoren, die wir zuerst noch kurz genauer beschreiben wollen.

Zuweisungsoperatoren

Zuweisungsoperatoren werden – wie der Name schon sagt – für die Zuweisung eines Werts zu einer Variablen verwendet. Sie sind als Abkürzung für Kombinationen aus arithmetischer Operation und Zuweisung zu verstehen. Die Anweisung `x += 3` entspricht beispielsweise der ausgeschriebenen Form `x = x + 3`. Oder `x /= 2` entspricht in der langen Version `x = x / 2`.

Modulooperator

Der **Modulooperator** dient zur Rückgabe des Rests einer Division. In Java als auch JavaFX ist der Modulooperator nicht nur für Ganzzahlen definiert (wie etwa in C/C++ oder den meisten anderen Techniken), sondern auch für Fließkommazahlen! Es ist einfach die natürliche Fortsetzung der Operation auf die Menge der Fließkommazahlen.

! ! ! ACHTUNG

Fließkommaarithmetik *ist allgemein extrem heikel wegen potenziellen* ***Rundungsproblemen*** *(das sind Ungenauigkeiten in der Berechnung). Gerade bei Modulorechnungen mit Gleitkommazahlen treten oft Rundungsprobleme auf. Diese Rundungsprobleme sind nicht spezifisch für das Java-Umfeld, sondern treten in fast allen Programmierumgebungen auf.*

Inkrement- und Dekrementoperator

Der **Inkrement**- und **Dekrementoperator** stammen explizit aus C und gehören zu den Operatoren, die man in einer extrem unverständlichen Weise einsetzen kann. Im Grunde ist der Operator sehr einfach und nützlich. Der einstellige Inkrementoperator dient einfach zum Erhöhen des Werts des Operanden um 1.

Das Problem ist nur, dass die **Reihenfolge** von Operand und Operator wichtig ist. Wenn der Operator **vor** dem Operanden steht, erfolgt die Erhöhung des Werts, **bevor** (!) der Wert dem Operanden zugewiesen wird. Steht er **hinter** dem Operanden, erfolgt die Erhöhung, **nachdem** (!) der Wert bereits zugewiesen wurde.

Wenn Sie also den Operator in komplexeren zusammengesetzten Ausdrücken verwenden, ist unter Umständen kaum nachvollziehbar, wie sich ein Wert entwickelt. Oder wüssten Sie auf Anhieb, welchen Wert die Variable x am Ende des nachfolgenden Quellcodesegments hat?

```
var x = 9;
x += x++ + ++x +x++ + x++ + ++x;
```

Listing 3.12: Syntaktisch ist das korrekt, aber absolut nicht durchschaubar

* * * TIPP

Man sollte den Operator auf jeden Fall nur als einzelne Anweisung verwenden. Zum Beispiel um in Schleifen einen Wert zu erhöhen.

Der einstellige Dekrementoperator erniedrigt vollkommen den Wert des Operanden um 1.

Ein vollständiges Beispiel mit arithmetischen Operatoren

Beispiel (*operatoren*) – beachten Sie die erklärenden Kommentare:

```
01 import java.lang.System;
02 import java.lang.Math;
03
04 // Variablendeklarationen mit Wertzuweisung
05 var x = 17;
06 var y = 3;
07 var z = 4;
08
09 // Ausgabe verschiedener Berechnungen mit arithemtischen Operatoren
10 System.out.println(x % y); // 17 % 3: 3 * 5 ist 15 - es bleibt der Rest 2
11 System.out.println(x % z); // 17 % 4: 4 * 4 ist 16 - es bleibt der Rest 1
12
13 x += 3; // 17 + 3 - auf den bisherigen Wert von x wird der Wert 3 addiert
14 System.out.println(x); // Ausgabe 20
15
16 x /=5; // 20 / 5 - der bisherige Wert von x wird durch den Wert 5 geteilt
17 System.out.println(x); // Ausgabe 4
18
19 x %= 3; // 4 % 3: 3 * 1 ist 3 - es bleibt der Rest 1
20 System.out.println(x); // Ausgabe 1
21
```

```
22 x++;
23 System.out.println(x); // Ausgabe 2
24
25 System.out.println(x++); // Ausgabe 2 - danach wird der Wert von x erhöht
26 System.out.println(x); // Ausgabe 3
27
28 // Modulo mit Gleitkommazahlen - Achtung Rundungsprobleme!
29 System.out.println(4.2 % 1.7);  // Ausgabe 0.8000000000000003
```

Listing 3.13: Verschiedene arithmetische Operatoren in einem praktischen Beispiel

Das ist die Ausgabe des Beispiels (beachten Sie die erklärenden Kommentare hinter den einzelnen Anweisungen):

2

1

20

4

1

2

2

3

0.8000000000000003

3.5.4 Weitere Operatoren

JavaFX stellt nun eine ganze Reihe an weiteren Operatoren zu Verfügung, die auch in den offiziellen Unterlagen von Sun nicht weiter klassifiziert werden. Insbesondere haben viele dieser Operatoren keine direkten Gegenstücke in Java. Dort werden sie allerdings entweder über Sprachkonstrukte oder vielfach Eigenschaften oder Methoden von Objekten abgebildet. Wir wollen die Operatoren hier vollständig auflisten, aber zum Teil werden die nun folgenden Operatoren in den nachfolgenden Abschnitten des Kapitels erst genauer erklärt. Sie benötigen schlicht und einfach für einige Anwendungen ein umfangreicheres Gerüst, was erst noch erarbeitet werden muss.

Operator in JavaFX	Bedeutung	Entsprechender Operator in Java
`sizeof`	Die Größe eines Arrays	Kein Gegenstück als Operator
`indexof`	Die ordinale Position in einer Liste oder einem Array	Kein Gegenstück als Operator
`if` *`(Bedingung)`* `then` *`(Anweisungen, wenn Bedingung erfüllt)`* `else` *`(Anweisungen, wenn Bedingung nicht erfüllt)`*	Ein Bedingungsausdruck beziehungsweise Bedingungsoperator. Bemerkenswert ist einmal, dass diese Token als Operator (mit drei Operanden – ein so genannter **triadischer Operator**) in JavaFX gewertet werden. Diese Kurzform tritt in anderer Syntax auch in Java auf (siehe die dritte Spalte), wo es einen entsprechenden Operator sowie eine äquivalente Entscheidungsstruktur als Anweisung gibt. Was aber noch auffallen sollte, ist das Schlüsselwort `then` im JavaFX-Operator. In einigen Sprachdialekten gibt es das Anweisungskonstrukt `if-then`. Diese Konstruktion wurde jedoch in Java und anderen C-basierten Sprachen für die `if`-Anweisung bewusst nicht gewählt, da das Schlüsselwort `then` schlicht und einfach überflüssig ist. Und bemerkenswerterweise wird auch die `if`-***Anweisung*** in JavaFX auf `then` verzichten. Für den Operator macht `then` aber trotzdem einen gewissen Sinn, um die Bedingung abzugrenzen.	*`(Bedingung)`* ? *`(Anweisungen, wenn Bedingung erfüllt)`* : *`(Anweisungen, wenn Bedingung nicht erfüllt)`*
`select`	Auswahl in einer Liste beziehungsweise einem Datenfeld	Kein Gegenstück als Operator
`foreach`	Auswahl sämtlicher Elemente in einer Liste beziehungsweise einem Datenfeld	Kein Gegenstück als Operator

Tabelle 3.6: Verschiedene Operatoren in JavaFX und ihre Gegenstücke in Java, falls diese vorhanden sind

Operator in JavaFX	Bedeutung	Entsprechender Operator in Java
`new`	Allokieren von Speicherplatz	`new`
`op()`	Aufruf einer Funktion/ Operation	Kein Gegenstück als Operator
`x.op()`	Aufruf einer Funktion/Operation als Mitglied eines Objekts. Das Gegenstück in Java ist in Bezug auf Objektmethoden zu sehen.	*`x.op()`*
`instanceof`	Überprüfung eines Typs eines Objekts	`instanceof`
`this`	Eine Selbstreferenz auf das aktuelle Objekt	`this`
`.`	Zugriff auf einen Bestandteil eines Objekts. In der zweiten Bedeutung ist es in JavaFX ein Zugriff auf den aktuellen Kontext.	Auch in Java erfolgt der Zugriff auf einen Bestandteil eines Objekts über den Punkt (`.`). In der Bedeutung als ein Zugriff auf den aktuellen Kontext gibt es kein Gegenstück als Operator in Java.
`bind [lazy]`	Inkrementelle (lazy) Evaluierung	Kein Gegenstück als Operator
:	Initialisierung einer Variablen beim Deklarieren	Kein Gegenstück als Operator. In Java erfolgt die Wertzuweisung grundsätzlich über den Gleichheitsoperator.
`[]`	Selektion in einem Datenfeld	`[]`
`format as`	Stringformatierung	Kein Gegenstück als Operator
<<>>	Französische Anführungszeichen, um eine beliebige Zeichensequenz als Bezeichner verwenden zu können.	Kein Gegenstück als Operator
`{}`	Ein Stringausdruck. Das bedeutet, der Ausdruck in dem String wird ausgewertet.	Kein Gegenstück als Operator
`(Ausdruck)`	Gruppierung	*`(Ausdruck)`*
`reverse`	Dreht eine Liste um.	Kein Gegenstück als Operator
`[number1,next..number2]`	Numerischer Bereich	Kein Gegenstück als Operator

Tabelle 3.6: Verschiedene Operatoren in JavaFX und ihre Gegenstücke in Java, falls diese vorhanden sind (Fortsetzung)

Einige der Operatoren können wir erst genauer besprechen, wenn wir wie gesagt erweiterte Techniken in JavaFX besprochen haben. Schauen wir uns dennoch zur Verdeutlichung wieder ein Beispiel mit verschiedenen Operatoren an (*operatoren2*). Die einzelnen Anweisungen sind umfangreich kommentiert und sollten damit nachvollziehbar sein:

```
01 import java.lang.System;
02 import java.lang.Math;
03
04 // Deklaration von verschiedenen Variablen
05 var x = 2;
06 var y = 4;
07 var a = true;
08 var b = false;
09
10 // Größe eines Arrays - die Ausgabe ist  2
11 System.out.println(sizeof [x, y]);
12
13 // Prüfung, ob die Variable a den Wert true hat. Da dem so ist,
14 // ist die Ausgabe 2. Die Anweisung hinter then wird ausgewählt.
15 System.out.println(if a then x else y);
16
17 // Ein Testwert q wird im Array gesucht. Bedingung ist, dass q > 3 ist
18 // Die Ausgabe ist deshalb 4 - der Wert von y
19 System.out.println(select q from q in [x, y] where q > 3);
20
21 // Ausgabe aller Werte in dem Array, die kleiner als der Wert 3 sind.
22 // Die Ausgabe ist 2 - der Wert der Variablen x
23 System.out.println(foreach(q in [x, y] where q < 3) q);
24
25 // Aufruf einer Klassenmethode - Das Maximum von x und y
26 // - die Ausgabe ist 4. Der Wert von y
27 System.out.println(Math.max(x, y));
28
29 // Für ein Stringobjekt wird eine Methode aufgerufen. Diese konvertiert
30 // den Inhalt in Großbuchstaben. Die Ausgabe ist ABC
31 System.out.println("abc".toUpperCase());
32
33 // Überprüfung, ob x eine Variable vom Typ Number ist.
34 // Die Ausgabe ist true
35 System.out.println(x instanceof Number);
```

Listing 3.14: Verschiedene Operatoren im Einsatz

Die Ausgabe sieht wie folgt aus:

2

2

4

2

4

ABC

true

3.5.5 Ausdrücke

Ausdrücke sind allgemein eines der wichtigsten Stilmittel in der Programmierung. Sie sind das, was einen Wert in einer Programmierung repräsentiert. Sie drücken einen Wert entweder direkt (durch ein Literal) oder durch eine Berechnung im weiteren Sinn aus. Ausdrücke können Konstanten, Variablen, Literale, Schlüsselwörter, Operatoren und andere Ausdrücke beinhalten Wir haben in unseren bisherigen Beispielen natürlich schon diverse Ausdrücke verwendet. In diesem Abschnitt wollen wir die Ausdrücke von JavaFX genauer untersuchen, denn die Thematik passt hervorragend zu Operatoren. Man kann Ausdrücke am einfachsten folgendermaßen definieren:

Ausdrücke sind das Ergebnis der Verbindung von Operanden und Operatoren über die syntaktischen Regeln der Sprache.

Ausdrücke werden also für die Durchführung von Operationen (Manipulationen) an Variablen oder Werten verwendet. Dabei sind Spezialfälle wie arithmetische Konstanten beziehungsweise Literale kein Widerspruch.

Die Bewertung von Ausdrücken

Ausdrücke kommen selbstverständlich auch über komplizierte Kombinationen von Operatoren und Operanden vor. Deshalb muss ein ausführendes System diese Kombinationen **bewerten**, also eine Reihenfolge festlegen, wie diese komplexeren Ausdrücke auszuwerten sind. Das ist in der menschlichen Logik nicht anders, etwa die Punkt-vor-Strich-Rechnung in der Mathematik. Überhaupt lässt sich die Bewertung von Ausdrücken in den meisten Skript- und Programmiersprachen durch die Auswertung von Ausdrücken in der Mathematik intuitiv herleiten.

Eine Bewertung von Ausdrücken basiert auf drei Grundbegriffen, die in nahezu allen Programmier- und Skriptsprachen analog umgesetzt werden:

- Alle arithmetischen Ausdrücke werden bei einer Verarbeitung von links nach rechts ausgewertet. Wenn Operatoren gleicher Priorität in einem Ausdruck mehr als einmal auftauchen – wie beispielsweise der +-Operator bei dem Ausdruck `1 + 2 + 3 + 4 + 5` –, dann wird der am weitesten links erscheinende Ausdruck (`1 + 2`) zuerst ausgewertet, gefolgt von dem rechts daneben usw. Dieses Verhalten nennt man **Operatorassoziativität**.
- Der so genannte **Operatorvorrang** bedeutet die Berücksichtigung der Priorität von Operatoren. Wenn Sie einen Ausdruck mit unterschiedlichen Operatoren haben, muss das System entscheiden, wie Ausdrücke ausgewertet werden. Dazu wird zuerst der gesamte Ausdruck analysiert. Anschließend wird der am höchsten priorisierte Operator zuerst. Runde Klammern haben die höchste Priorität, weshalb jeder Ausdruck in Klammern zuerst ausgewertet wird.
- Als dritten Begriff muss man die so genannte **Bewertungsreihenfolge** beachten. Die Bewertungsreihenfolge bewertet im Gegensatz zum Operatorvorrang die Operanden. Die Bewertungsreihenfolge legt fest, welche Operatoren in einem Ausdruck zuerst benutzt werden und welche Operanden zu welchen Operatoren gehören. Außerdem dienen die Regeln für die Bewertungsreihenfolge zur Festlegung, wann Operanden ausgewertet werden.

3.5.6 Stringliterale und Stringausdrücke in JavaFX

Wie schon erwähnt, können Sie in JavaFX eine Zeichenkette sowohl in einfache als auch doppelte Hochkommata einschließen. Und wie auch beispielsweise in PHP werden Zeichenketten in doppelten und einfachen Hochkommata vom System bei gewissen Situationen unterschiedlich behandelt.

Auswertung von Ausdrücken in Strings

Wenn Sie beispielsweise den Bezeichner einer Variablen in eine Zeichenkette mit doppelten Hochkommata notieren und diesen in geschweifte Klammern `{}` einschließen, wird der Ausdruck **ausgewertet**. Sie erhalten in dem resultierenden String den Wert der eingeschlossenen Variablen beziehungsweise eines Ausdrucks. Dabei sind Sie für die Art der ausgewerteten Variable nicht auf Zeichenketten beschränkt. Bei einer Zeichenkette in einfachen Hochkommata finden Sie dieses Verhalten nicht[89].

89 In PHP kennt man ein ähnliches Verhalten.

Machen wir uns das in einem Beispiel deutlich (*Zeichenketten*):

```
01 import java.lang.System;
02
03 var name = 'Trillian';
04 var wert = 5;
05 var s1 = "Hallo {name}"; // Zeichenkette mit doppelten Hochkommata
06 var s2 = 'Hallo {name}'; // Zeichenkette mit einfachen Hochkommata
07 var s3 = "Hallo {wert}"; // Zeichenkette mit doppelten Hochkommata
08
09 System.out.println(s1); // Hallo Trillian
10 System.out.println(s2); // Hallo {name}
11 System.out.println(s3); // Hallo 5
```

Listing 3.15: Auswerten von Variablen in Strings

Die Ausgabe ist folgende:

Hallo Trillian

Hallo {name}

Hallo 5

* * * TIPP

*Diese Möglichkeit zum Auswerten von Ausdrücken in Zeichenketten erleichtert für Einsteiger den Umgang mit Variablenwerten und Textausgaben und ist deshalb verführerisch. Ich denke aber, viele erfahrene Umsteiger aus anderen Sprachen werden dennoch – am Anfang – die traditionelle Version des Zusammensetzens einer Zeichenkette vorziehen. Und sei es nur, um sich keinen Programmierstil anzugewöhnen, der in anderen Sprachen – insbesondere Java – nicht funktioniert. Das Beispiel von eben könnte man auch so schreiben, was zwar auf den ersten Blick etwas umständlicher erscheint, von der Struktur dann aber auch in anderen Sprachen funktioniert (*zeichenketten2*):*

```
// String-Variable wird mit concat() an String angehängt var s1 =
  "Hallo ".concat(name);
var wert = 5;
// Numbervariable wird mit toString()in String umgewandelt und mit
// concat() an String angehängt
var s3 = "Hallo ".concat(wert.toString());
```

Listing 3.16: Die »saubere« Variante zur Verwertung von Variablenwerten in Verbindung mit einem String

Dennoch habe ich mich – nachdem ich am Anfang diese Auswertung von Ausdrücken in Strings gar nicht gemocht habe – mehr und mehr damit anfreunden können. Und zwar je mehr ich insbesondere dynamische grafische Oberflächen mit JavaFX erstellt habe. Hier ist die Möglichkeit der Auswertung von Ausdrücken innerhalb eines Strings für einige Vorgänge schlicht und einfach genial.

In JavaFX kann der eingeschlossene Ausdruck selbst wieder Strings in doppelten Hochkommata enthalten, worin sogar weitere Ausdrücke eingeschlossen sein können.

> > > HINWEIS

So etwas ist in mir bekannten vergleichbaren anderen Sprachen nur mit ausdrücklicher Markierung der eingeschlossenen Hochkommata möglich.

Das nachfolgende Beispiel funktioniert in JavaFX:

```
01 import java.lang.System;
02
03 var antwort = true;
04 var s = "Die Antwort ist {if antwort then "Ja" else "Nein"}.";
05 // s hat nun den Wert 'Die Antwort ist Ja'
06 System.out.println(s);
```

Listing 3.17: Verwenden von eingeschlossenen doppelten Hochkommata

Nach dem Import in Zeile 1 wird in Zeile 3 eine Boolesche Variable `antwort` eingeführt und mit dem Wert `true` versehen (eine indirekte Festlegung des Datentyps über den Datentyp des Literals).

In Zeile 4 wird ein Ausdruck innerhalb eines Strings ausgewertet – in diesem Fall der Bedingungsoperator. Da der Wert der Bedingung in Zeile 3 als `true` festgelegt wird, wird die Anweisung hinter `then` ausgewählt und der Wert in den umgebenden String eingefügt. Dies ist der String mit dem Inhalt `"Ja"`. Also ein Stringliteral, das in doppelte Hochkommata eingeschlossen ist.

Das ist die Ausgabe des Beispiels:

Die Antwort ist Ja.

Zeilenumbrüche in Strings

Im Gegensatz zu Java können in JavaFX Zeichenketten Zeilenumbrüche enthalten, wenn sie in doppelte Hochkommata eingeschlossen sind.

Beispiel:

```
01 var s = "Die
02 Antwort ist
03 42";
```

Listing 3.18: Eine Zeichenkette geht über mehrere Zeilen

> > > HINWEIS

Ehrlich gesagt hat es mir beim ersten Kontakt mit der Möglichkeit, eine Zeichenkette so auf mehrere Zeilen aufzuteilen, regelrecht gegraut. Aber auch die Auswertung von Ausdrücken in Strings sowie das Einschließen von Hochkommata gleichen Typs in einen String ohne explizite Markierung hat mich ganz kribbelig gemacht. Solche Sachen gehören in fast allen von mir sonst verwendeten Programmiersprachen[90] zu den klassischen Anfängerfehlern, die ich meinen Schulungsteilnehmern austreiben muss. Und nun das – eine Sprache, in der so etwas keinen Fehler mehr darstellt.

Aber um es noch einmal zu erwähnen – mein Aha-Erlebnis war die Erstellung von grafischen Oberflächen mit JavaFX. Hier eröffnen diese ungewohnten Verhaltensweisen beim Erstellen von Quellcode einfach geniale Möglichkeiten und vereinfachen die Erstellung immens.

3.6 Anweisungen

Anweisungen sind in einer Programmier- oder Skriptsprache solche Syntaxstrukturen, die bestimmte Dinge aktiv ausführen. Anweisungen werden in der Regel einfach der Reihe nach ausgeführt. Ausnahmen sind Kontrollflussanweisungen oder Ausnahmeanweisungen. Sie werden aufgrund eines besonderen Effekts ausgeführt.

Grundsätzlich unterteilt man Anweisungen nach ihrer Art.

> > > HINWEIS

Beachten Sie, dass wir die meisten Anweisungsformen in unseren bisherigen Beispielen schon gesehen haben.

3.6.1 Blockanweisung

Eine **Blockanweisung** ist einfach eine Zusammenfassung von mehreren Anweisungen zu einer Blockstruktur. In Java und JavaFX erledigen das die geschweiften Klammern, wie wir schon häufiger gesehen haben. So weit lesen Sie hier auch nichts Neues.

Aber das ist neu. Ein Anweisungsblock hat in Java und JavaFX seinen eigenen Geltungsbereich für die in ihm enthaltenen Anweisungen. Das bedeutet, dass in einigen Blockstrukturen lokale Variablen in diesem Block deklariert werden können, die außerhalb dieses Blocks nicht verfügbar sind und deren Existenz erlischt, wenn der Block ausgeführt wurde – beispielsweise so genannte **schleifenlokale Variablen**, die nur innerhalb einer Schleife[91] zur Verfügung stehen.

90 Mit Ausnahme von PHP – auch da kann man fast genauso viele schlimme ;-) Sachen machen.

91 Sollten Sie noch nie etwas von Schleifen gehört haben – stellen Sie sich diese bis auf weiteres einfach als eine Wiederholung von Anweisungen vor.

Schauen wir uns dazu ein Beispiel (*blockanweisungen*) an, das auf Schleifen vorgreift (die werden aber unmittelbar in der Folge erläutert):

```
01 import java.lang.System;
02
03 var x = 1;
04 for(i in [0..5]) {
05   var x = i;
06   System.out.println(x);
07 }
08 System.out.println(x);
```

Listing 3.19: Ein Block mit einer schleifenlokalen Variablen

In Zeile 3 wird eine global gültige Variable `x` eingeführt. Diese bekommt direkt bei der Deklaration den numerischen Wert 1 zugewiesen.

In der folgenden Schleife wird ebenfalls mit einer Variable mit dem gleichen Bezeichner `x` gearbeitet. Der zugehörige Block erstreckt sich von Zeile 4 bis 7. Diese schleifenlokale Variable bekommt in Zeile 5 jeweils den Wert der Zählvariablen `i` zugewiesen.

In Zeile 6 wird der jeweilige Wert von `x` ausgegeben (von 0 bis 5). Das ist der Wert, der in der Schleife zugewiesen wird.

In Zeile 8 wird anschließend der Wert der global definierten Variable `x` ausgegeben. Hätte die Variable `x` in dem Block keinen eigenen Gültigkeitsbereich, müsste der Wert dieser Variablen durch die Zuweisungen in der Schleife verändert worden sein. Die Ausgabe beweist aber, dass dem nicht so ist. Der Wert der global definierten Variable `x` ist weiter 1:

0

1

2

3

4

5

1

3.6.2 Deklarationsanweisung

Eine **Deklarationsanweisung** ist eine Anweisung zur Einführung einer Variabeln, einer Funktion, einer Operation, einer Klasse oder eines Objekts. Dazu soll an dieser Stelle nicht mehr ausgeführt werden, da wir Deklarationsanweisungen bei den jeweiligen Themen direkt besprechen beziehungsweise auch schon besprochen haben.

3.6.3 Ausdrucksanweisung

Ausdrucksanweisungen sind der offizielle Name für die folgenden Anweisungsarten, die wir an anderen Stellen genauer behandeln beziehungsweise schon behandelt haben:

- Zuordnungen
- Inkrement
- Dekrement
- Aufrufe von Methoden beziehungsweise Funktionen und Operationen
- Zuweisungen

Alle Ausdrucksanweisungen müssen in JavaFX wie in Java mit einem Semikolon beendet werden und eine Ausdrucksanweisung wird immer vollständig durchgeführt, bevor die nächste Anweisung ausgeführt wird.

3.6.4 Leere Anweisung

Es gibt in Java auch eine **leere Anweisung**. Sie besteht nur aus einem Semikolon und dient als Platzhalter. In JavaFX gibt es diese leere Anweisung nicht! Mit anderen Worten: Ein einzelnes Semikolon, das nicht eine Anweisung beendet, wird in JavaFX als Fehler gewertet.

3.6.5 Auswahlanweisung

Eine **Auswahlanweisung** ist eine **Kontrollflussanweisung**. Mit einer Auswahlanweisung sucht man einen von mehreren möglichen Kontrollflüssen eines Programms beziehungsweise Skripts heraus.

Sowohl Java als auch JavaFX unterstützt verschiedene Auswahlanweisungen. Allerdings sind diverse Unterschiede zu beachten. Insbesondere kennt JavaFX beispielsweise keine switch-case-Anweisung. Das ist jedoch kein schlimmer Verlust, denn mit den verbleibenden `if-` beziehungsweise `if-else`-Anweisungen kann man alles machen, was zur Steuerung des Programmflusses notwendig ist.

Die if- und die if-else-Anweisung

Eine **if-Anweisung** testet eine Boolesche Variable oder einen Ausdruck. Wenn die Boolesche Variable oder der Ausdruck den Wert `true` hat, wird die nachstehende Anweisung oder der nachfolgende Anweisungsblock ausgeführt. Ansonsten wird die nachstehende Anweisung oder der nachstehende Anweisungsblock ignoriert und mit dem folgenden Block beziehungsweise der folgenden Anweisung fortgefahren.

Eng verwandt ist die **if-else-Anweisung**, die genau genommen nur eine Erweiterung der if-Anweisung ist. Sie hat nur noch einen zusätzlichen `else`-Teil. Dieser `else`-Teil wird dann ausgeführt, wenn der Boolesche Test im `if`-Teil der Anweisung den Wert `false` ergibt.

! ! ! ACHTUNG

*Im Grunde unterscheiden sich Java und JavaFX nicht bezüglich der `if`- beziehungsweise `if-else`-Anweisung. Nur müssen Sie in JavaFX die Anweisungen **zwingend** in Blöcke notieren. In Java kann man eine einzelne Anweisung auch ohne umgebende geschweifte Klammern notieren, obwohl davon dringend abzuraten ist. Zu leicht vergisst man bei einer späteren Erweiterung um weitere Anweisungen die dann notwendigen Klammern. So gesehen ist der Zwang in JavaFX eine positive Entwicklung.*

Beispiel (*ifanweisung*):

```
01 import java.lang.System;
02
03 var x = 4;
04 if(x == 3) {
05   System.out.println("a");
06 }
07 else {
08   System.out.println("b");
09 }
```

Listing 3.20: Eine einfache if-else-Anweisung

Die Ausgabe des Beispiels ist *b*.

3.6.6 Schleifen

Schleifen dienen allgemein der Wiederholung von Anweisungen. Auch bei Schleifen muss man zwischen Java und JavaFX diverse Unterschiede beachten.

Die while-Anweisung

Die **while-Anweisung** sieht schematisch wie folgt aus:

```
while(Bedingung) Block
```

Listing 3.21: Schema für eine while-Schleife

Wenn der in der Bedingung überprüfte Ausdruck nicht von Anfang an den Wert `true` liefert, wird der Block niemals ausgeführt. Man nennt dies eine **kopfgesteuerte** oder **abweisende** Iterationsanweisung.

Wenn der Test in der Bedingung hingegen `true` liefert, dann wird dieser Codeblock so lange wiederholt, bis der Test nicht mehr `true` liefert. Alternativ kann durch eine **Sprunganweisung** (siehe Seite 125) die Kontrolle an eine Anweisung außerhalb der Schleife weitergegeben werden.

! ! ! ACHTUNG

Im Grunde unterscheiden sich Java und JavaFX auch nicht bezüglich der `while`-Anweisung. Nur müssen Sie wie bei der `if`-Anweisung in JavaFX die Anweisung nach der Bedingung zwingend in einen Block notieren. In Java kann man eine einzelne Anweisung auch ohne umgebende geschweifte Klammern notieren, obwohl davon – wie bei der `if`-Anweisung zu ausgeführt – dringend abzuraten ist.

Beispiel (*whileanweisung*):

```
01 import java.lang.System;
02
03 var i = 0;
04 while (i < 10) {
05   System.out.println("i = {i}");
06   i++;
07 }
```

Listing 3.22: Eine while-Schleife

Die Zählvariable `i` wird in Zeile 3 eingeführt und mit dem Wert 0 initialisiert.

In Zeile 4 wird überprüft, ob der Wert der Zählvariablen kleiner als 10 ist (`while (i < 10) {`). Wenn das der Fall ist, wird der Wert von `i` ausgeben. Dabei wird der Wert von `i` in einem String in geschweifte Klammern notiert und damit ausgewertet.

In Zeile 6 wird der Wert der Zählvariablen mit dem Inkrementoperator erhöht (`i++;`).

Das ist die Ausgabe:

i = 0

i = 1

i = 2

i = 3

i = 4

i = 5

i = 6

i = 7

i = 8

i = 9

! ! ! ACHTUNG

*Allgemein sind Schleifen Kandidaten für so genannte **Endlosschleifen**. Das sind Schleifen, die nie beendet werden und immer wieder die darin notierten Anweisungen ausführen. Eine `while`-Schleife ist ein besonders »guter« Kandidat für eine solche – ungewollte – Endlosschleife. Leicht übersieht man im Inneren der Schleife eine Situation zu schaffen, in der die Bedingung zum Durchlaufen der Schleife nicht mehr erfüllt ist, oder durch einen Sprung die Schleife zu verlassen. Dann hängen Sie in einer Endlosschleife.*

Die for-Anweisung

Diese Iterationsanweisung ist kopfgesteuert und fasst bereits im Schleifenkopf alle relevanten Anweisungen zur Steuerung der Schleife zusammen. Deshalb ist diese Iterationsanweisung bei den meisten Programmierern die beliebteste Schleifenform. Nun hat Sun für die `for`-Anweisung in JavaFX aber nicht die Syntax gewählt, wie sie in Java und allen C-basierenden Sprachen verwendet wird. Diese steht auch gar nicht mehr zur Verfügung!

Stattdessen verwendet man in JavaFX eine Syntax, die sonst eigentlich nur in einer Spezialform der `for`-Anweisung – der `foreach`-Anweisung – Verwendung findet. Der Kopf der `for`-Anweisung in JavaFX verwendet sogar die gleiche Syntax wie der `for-each`-Operator. Angeblich soll diese Version der Syntax einer `for`-Schleife für Einsteiger leichter zu verstehen sein[92], aber das ist wahrscheinlich nicht der wirkliche Grund für diesen Umstieg gegenüber der Java-Syntax.

Diese Form der Schleife minimiert die Gefahr von ungewollten Endlosschleifen.

Eine einfache `for`-Anweisung sieht in JavaFX vom Schema her wie folgt aus:

```
for (Zählvariable in [Beginn Wertebereich .. Ende Wertebereich]) Block
```

Listing 3.23: Schema der Syntax für eine for-Anweisung

Beim ersten Schleifendurchlauf wird die Zählvariable auf den Anfangswert des Wertebereichs gesetzt. Vor jedem weiteren Schleifendurchlauf wird überprüft, ob sich die Zählvariable im dem angegebenen Wertebereich befindet. Während eines Schleifendurchlaufs wird die Zählvariable automatisch um den Wert 1 erhöht, wenn der Endwert größer als der Startwert ist. Ist der Endwert jedoch kleiner als der Startwert, wird die Zählvariable während jedes Schleifendurchlaufs automatisch um den Wert 1 erniedrigt. Das entspricht also einem Rückwärtszählen.

Beispiel (den Quelltext als auch die nachfolgenden Ausführungen zu der `for`-Schleife finden Sie im Ordner *foranweisung*):

92 Ich persönlich ziehe aber die C-Syntax in der Regel vor.

```
01 import java.lang.System;
02
03 // Ausgabe von 0 bis 5
04 for (i in [0..5]) {
05   System.out.println("i = {i}");
06 }
07
08 // Ausgabe von 3 bis 5
09 for (i in [3..5]) {
10   System.out.println("i = {i}");
11 }
12
13 // Ausgabe von 5 bis 2
14 for (i in [5..2]) {
15   System.out.println("i = {i}");
16 }
```

Listing 3.24: Eine einfache for-Anweisung

Die erste Schleife wird sechs Mal durchlaufen und gibt dabei jeweils den aktuellen Wert der Zählvariablen aus. Beachten Sie in Zeile 15 die Auswertung der Variable `i` über `{i}` innerhalb des Strings. Die zweite Schleife beginnt mit dem Startwert 3 und die dritte Schleife erniedrigt jeweils die Zählvariable.

Die `for`-Anweisung in JavaFX wurde nun um ein paar mehr oder weniger nützliche Tricks erweitert.

In der klassischen Version einer `for`-Anweisung kann man die **Schrittweite** (also die Größe der Werterhöhung der Zählvariablen) flexibel verändern. Es gibt zwar kaum einen sinnvollen Grund, warum man das tun sollte[93], aber in der Version von JavaFX hat man dennoch diese Möglichkeit vorgesehen.

Dazu kommt aber eine doch recht eigenwillige Syntax zum Einsatz, die so auch allgemein bei Datenfeldern verwendet werden kann (siehe Seite 136). Am Beginn des Wertebereichs gibt man zwei Zahlen an, die man mit einem Komma trennt. Die Differenz zwischen den Zahlen gibt die Schrittweite an.

Beispiel:

```
01 import java.lang.System;
02
03 for (i in [1,4..10]) {
04   System.out.println("i = {i}");
05 }
```

Listing 3.25: Eine andere Schrittweite für die Schleife

93 Beachten Sie dazu den nachfolgenden Hinweis.

Das Beispiel gibt die Werte 1, 4, 7 und 10 aus, weil die Schrittweite 3 ist (4 – 1).

Ein weiterer Trick in einer `for`-Anweisung ist eine genauere Angabe, wie die Zählvariable sich verhalten soll. Dazu kommen so genannte **Filter** zum Einsatz, die der SQL-Syntax entsprechen und analog bei Arrays Verwendung finden (siehe dazu Seite 133)[94]. Sie werden mit dem Schlüsselwort `where` eingeleitet.

Beachten Sie das nachfolgende Beispiel:

```
01 import java.lang.System;
02
03 // Ausgabe aller geraden Zahlen über einen Filter
04 for (i in [0..10] where i % 2 == 0) {
05   System.out.println("i = {i}");
06 }
```

Listing 3.26: Einsatz eines Filters

Mit diesem Beispiel werden alle geraden Zahlen von 0 bis 10 ausgegeben.

> > > HINWEIS

*So interessant die Abarten eine `for`-Schleife auch erscheinen mögen – man kommt bei einer `for`-Anweisung (und jeder Schleife) **immer** mit der Schrittweite 1 ohne weitere Angaben aus. Selbst wenn man auf den ersten Blick vielleicht eine andere Schrittweite benötigt (etwa um nur alle geraden Zahlen zu durchlaufen), kann man durch einfaches Anpassen der Anweisungen im Körper der Schleife immer auf Basis einer Schrittweite 1 agieren. Auch die Möglichkeit des Reduzierens der Zählvariablen oder der Einsatz von Filtern für die Zählvariable ist deshalb im Grunde vollkommen überflüssig.*

Solche »Tricks« widersprechen sogar modernen Tendenzen in der Programmierung, die Programmierer zu einem »langweiligen« Programmierstil zu bewegen versuchen. Langweilig wäre beispielsweise, wenn alle `for`-Schleifen im Header exakt gleich aussehen (bis auf den wirklichen Endwert). Damit wird die Lesbarkeit und Wartbarkeit von Quellcode massiv erhöht, was letztendlich in besseren, stabileren Programmen mündet.

*Etwas anderes ist es, wenn man die Sache aus Sicht eines Arrays (*siehe Seite 130*) betrachtet. Die Angabe in den eckigen Klammern bei einer `for`-Schleife ist ein Array. Und hier erleichtert JavaFX mit diesen Möglichkeiten den Umgang doch erheblich. Wie schon an verschiedenen anderen Stellen sollten sich meines Erachtens erfahrene Programmierer in Geduld üben und nicht gleich Ansätze von JavaFX verteufeln, die scheinbar professionellen Regeln widersprechen. Wenn man JavaFX verinnerlicht hat, zeigen sich an vielen Stellen sehr intelligente Ideen hinter scheinbar unprofessionellen Ansätzen.*

94 Genau genommen handelt es sich bei der Angabe des Wertebereichs einer for-Anweisung auch um ein Array.

Sie können in einer `for`-Anweisung nun auch mehrere Zählvariablen parallel verwenden. Die verschiedenen Zählvariablen werden dabei getrennt erhöht. Dieses Konstrukt ist ein Ersatz für **verschachtelte for-Schleifen** und aus meiner Sicht eine der nützlichsten Erweiterungen einer einfachen `for`-Schleife.

Beispiel:

```
01 import java.lang.System;
02
03 for (i in [0..2], j in [0..2]) {
04      System.out.println("i = {i}");
05      System.out.println("j = {j}");
06 }
```

Listing 3.27: Eine spezielle JavaFX-Variante von verschachtelten for-Schleifen

In dem Beispiel wird zuerst die Zählvariable `i` auf den Wert 0 gesetzt und anschließend der Wertebereich für `j` durchlaufen. Dann wird `i` erhöht und erneut der Wertebereich von `j` durchlaufen usw.

Die Ausgabe ist das:

i = 0

j = 0

i = 0

j = 1

i = 0

j = 2

i = 1

j = 0

i = 1

j = 1

i = 1

j = 2

i = 2

j = 0

i = 2

j = 1

i = 2

j = 2

Die do- und die do-later-Anweisung

Die `do`-Anweisung (oder auch `do-while`-Anweisung genannt) in Java testet eine Boolesche Variable oder einen Ausdruck – aber erst am Ende eines Blocks, weshalb man von einer annehmenden oder fußgesteuerten Iterationsanweisung spricht. Damit wird der Codeblock innerhalb der `do`-Anweisung auf jeden Fall mindestens einmal ausgeführt.

Das Tückische in JavaFX ist nun aber, dass die `do`-Anweisung dort **nicht** diese Funktionalität hat, sondern sowohl eine andere Syntax als auch eine gänzlich anders gelagerte Anwendung. Es ist keine Schleife im klassischen Sinn und die Thematik in Verbindung mit der `do`-Anweisung ist alles andere als trivial. Sie wird im Umfeld des so genannten Multithreadings eingesetzt. Ebenso die verwandte Anweisung `do later`. Wir werden die `do`-Anweisung, wie sie in JavaFX realisiert ist, und die `do-later`-Anweisung in dem Kapitel 7 zu den erweiterten JavaFX-Techniken besprechen.

3.6.7 Sprunganweisung

So genannte **Sprunganweisungen** geben in einem Programm oder Skript den Programmfluss bei ihrem Aufruf unmittelbar entweder an den Anfang oder das Ende des derzeitigen Blocks weiter. Java und JavaFX kennen vier Arten von Sprunganweisungen, die fast gleich implementiert sind:

- `break`
- `continue`
- `return`
- `throw`

Die `break`- und die `return`-Anweisung haben wir im Laufe des Buchs ja bereits mehrfach behandelt. Aber dennoch gibt es auch dazu noch einige explizite Feinheiten sowie auch Unterschiede zu Java zu beachten.

Die return-Anweisung

Mit der `return`-Anweisung verlassen Sie eine Funktion oder Operation und liefern einen Rückgabewert, wenn Sie nach dem Schlüsselwort `return` einen solchen Wert (als Literal oder berechnet) notieren.

Letzteres bedeutet, Sie können auch die `return`-Anweisung notieren, **ohne** danach einen Rückgabewert zu schreiben. Das geht zumindest bei einer Operation beziehungsweise Prozeduren. Sie notieren dann nur direkt ein Semikolon nach dem Schlüsselwort `return`. Allerdings können Sie auch bei einer Operation einen Rückgabewert notieren (also einen Wert hinter `return` notieren).

! ! ! ACHTUNG

*Der Aufruf von `return` **ohne** Rückgabewert geht nicht bei Funktionen. Hier müssen Sie **immer** einen Rückgabewert notieren.*

Beispiel (*returnanweisung*):

```
01 import java.lang.System;
02
03 function add(x, y) {
04   // Rückgabewert der Funktion
05   return x + y;
06 }
07 operation add2(x, y) {
08   System.out.println(x + y);
09   // Aufruf von return oder Rückgabewert - das geht
10   // in einer Operation
11   return ;
12 }
13
14 System.out.println(add(7,1));
15 add2(4,5);
```

Listing 3.28: Die Verwendung von return in einer Funktion und einer Prozedur

! ! ! ACHTUNG

Es ist leider durch eine unglückliche Kodierung des Quelltexts möglich, Codezeilen zu schreiben, die nie erreicht werden können. Das kann passieren, wenn Sie in einer Funktion oder Operation nach einer `return`-Anweisung, die auf jeden Fall ausgeführt wird (also nicht in Abhängigkeit von einer Bedingung), weitere Anweisungen notieren.

*Dies nennt man dann **unerreichbaren Code** (**unreachable Code**). Solcher Code ist zwar eigentlich nicht schädlich in dem Sinne, dass er etwas Falsches tut (im Gegenteil – er wird ja gar nicht ausgeführt), aber wenn man sich darauf verlässt, dass bestimmte Codezeilen ausgeführt werden, und sie werden einfach nicht erreicht, kann der Schaden mindestens genauso groß sein.*

Der Java-Compiler bemerkt solche Konstruktionen mit unerreichbarem Code glücklicherweise fast immer und erzeugt einen Fehler zur Kompilierzeit. Soweit die gute Nachricht!

*Die schlechte Nachricht – **der JavaFX-Interpreter bemerkt unerreichbaren Code nicht**! Zumindest nicht in der Version, die dem Buch zugrunde liegt. Und das führt in der Tat dazu, dass eine JavaFX-Applikation mit unerreichbarem Code zum Laufen gebracht werden kann.*

Beispiel:

```
01 import java.lang.System;
02
03 operation add2(x, y) {
04   System.out.println(x + y);
05   return ;
06   // Achtung - unerreichbarer Code
07   System.out.println(x + y);
08 }
09 add2(4,5);
```

Listing 3.29: Das Beispiel funktioniert trotz unerreichbarem Code

Zeile 5 bewirkt, dass die Operation verlassen wird. Zeile 7 wird niemals erreicht. Das ist unerreichbarer Code.

Es ist bedauerlich, dass (derzeit) in JavaFX einer der intelligentesten Sicherheits- und Stabilitätsmechanismen nicht funktioniert. Die Verantwortung für die Vermeidung solcher Fehlersituationen liegt also wieder rein beim Programmierer.

Die break-Anweisung

Die `break`-Anweisung in JavaFX verhält sich wie die gleiche Anweisung in Java. Nur werden keine Sprünge an Labels (bezeichnete Sprungmarken) unterstützt[95]. Wie in Java muss die `break`-Anweisung im Körper einer Schleife verwendet werden. Der Programmfluss verlässt beim Auftreten der `break`-Anweisung unmittelbar die Schleife.

Beispiel (*breakanweisung*):

```
01 import java.lang.System;
02
03 operation breaktest(y) {
04     System.out.println("Start");
05     for (i in [0..5]) {
06     if (i > y) {
07       System.out.println("Abbruch vor Ende des Wertebereichs");
08       break;
09     }
10     System.out.println(i);
11   }
12   System.out.println("Ende");
13 }
14 breaktest(7);
15 breaktest(3);
```

Listing 3.30: Die Verwendung von break

95 Das ist aber keine wesentliche Einschränkung, denn Sprünge an Label werden nicht viel eingesetzt. Zumal solche gesteuerten Sprungadressen an Zeiten unseligen Spaghetticodes mittels des Goto-Befehls und Konsorten erinnert (obgleich benannte Sprungadressen in Java beileibe nicht so schlimm sind).

In dem Beispiel wird die `break`-Anweisung aufgerufen, wenn die Zählvariable `i` der `for`-Schleife größer als der Wert der Variable `y` ist, der als Übergabewert beim Aufruf der Prozedur angegeben wird. Das bedeutet, wenn der Wert des Parameters größer als 5 ist, wird die Schleife nicht zu Ende abgearbeitet, sondern vorher mit `break` abgebrochen. Die Ausgabe des Aufrufs in Zeile 15 zeigt das.

Das ist die Ausgabe des Beispiels:

Start

0

1

2

3

4

5

Ende

Start

0

1

2

3

Abbruch vor Ende des Wertebereichs

Ende

! ! ! ACHTUNG

Leider kann man auch mit der `break`-Anweisung in JavaFX unerreichbaren Code erzeugen.

Die continue-Anweisung

Die `continue`-Anweisung kommt nur in Zusammenhang mit Iterationsanweisungen zum Einsatz und bricht dort einen **Schleifendurchlauf** ab. Durch `continue` wird also im Gegensatz zu `break` oder `return` nicht die gesamte Schleife abgebrochen, sondern es wird nur der aktuelle Schleifendurchlauf unterbrochen und zum Anfang der Schleife zurückgekehrt. Die `continue`-Anweisung in JavaFX verhält sich wie die gleiche Anweisung in Java. Nur sind auch hier keine Sprünge an Labels möglich.

Beispiel (*continueanweisung*):

```
01 import java.lang.System;
02
03 operation continuetest() {
04     System.out.println("Ausgabe ganze Zahlen");
05     for (i in [0..20]) {
06     if (i % 2 <> 0) {
07       System.out.println("Ungerade Zahl");
08       continue;
09     }
10     System.out.println(i);
11   }
12 }
13
14 continuetest();
```

Listing 3.31: Die Ausgabe von ganzen Zahlen mit continue

In der Zeile 6 wird überprüft, ob die Zählvariable `i` modulo dem Wert 2 von 0 verschieden ist. Wenn dem so ist, muss eine ungerade Zahl vorliegen. Dann wird ein reiner Text ausgegeben und mit `continue` der nächste Schleifendurchlauf gestartet.

Das ist die Ausgabe des Beispiels:

Ausgabe ganze Zahlen

0

Ungerade Zahl

2

Ungerade Zahl

4

Ungerade Zahl

6

Ungerade Zahl

8

Ungerade Zahl

10

Ungerade Zahl

12

Ungerade Zahl

14

Ungerade Zahl

16

Ungerade Zahl

18

Ungerade Zahl

20

Die throw-Anweisung

Die `throw`-Anweisung ist wesentlicher Bestandteil des Konzepts der Ausnahmebehandlung in JavaFX. Wir werden sie explizit in Kapitel 7 bei den erweiterten JavaFX-Techniken besprechen.

3.6.8 Die try-Anweisung

Die `try`-Anweisung ist wie die Anweisung `throw` wesentlicher Bestandteil des Konzepts der Ausnahmebehandlung. Wir werden auch diese explizit in Kapitel 7 zu den erweiterten JavaFX-Techniken besprechen.

3.7 Arrays und der Operator []

Kommen wir nun zu einem Thema, das in JavaFX aus meiner Sicht für eine Programmier- beziehungsweise Skriptsprache teilweise recht ungewöhnlich – und sehr effektiv und intelligent[96] – umgesetzt wurde. An dieser Stelle berühren wir eine der wichtigsten Variablenstrukturen in vielen Programmiersprachen – die so genannten **Arrays** oder **Datenfelder**. Diese Strukturen werden in JavaFX (wie auch Java und vielen anderen Programmiersprachen) mit den eckigen Klammern gekennzeichnet.

Ungewöhnlich ist in JavaFX jedoch die Verwendung der eckigen Klammern bei einer Initialisierung, wie Sie sehen werden. Und auch sonst hat das Array-Verfahren in JavaFX nur auf den ersten Blick große Ähnlichkeit mit dem Vorgehen bei Datenfeldern oder Arrays in Java. Es werden erfahrenen Programmierern aus Java als auch anderen Sprachen einige interessante Überraschungen bevorstehen.

96 Wenn man sich daran gewöhnt hat!

3.7.1 Was sind Arrays?

Arrays beziehungsweise Datenfelder bezeichnen allgemein eine sequenzielle Sammlung von Inhalten gleichen Typs, die alle über einen gemeinsamen Bezeichner und einen in eckigen Klammern notierten Index (in Java als auch JavaFX bei 0 beginnend) angesprochen werden können.

Beispiel:

```
zahlen [1];
```

Listing 3.32: Zugriff auf den zweiten Wert eines Arrays

Datenfelder sind beispielsweise immer dann von großem Nutzen, wenn eine Reihe von gleichartigen oder logisch zusammenfassbaren Informationen gespeichert werden sollen. Ein Hauptvorteil von Arrays ist, dass der Zugriff auf die einzelnen Einträge im Array über den numerischen Index erfolgen kann. Das kann man beispielsweise in Schleifen gut ausnutzen.

3.7.2 Arrays in JavaFX

Trotz der unbestreitbaren Ähnlichkeit aller Array-Techniken im Allgemeinen und der Verwandtschaft von Java und JavaFX im Besonderen implementiert JavaFX Arrays teilweise sehr ungewöhnlich. Es handelt sich insbesondere bei Arrays in JavaFX nicht um die gleichen Arrays, wie sie in Java vorkommen.

Zum Ersten werden in JavaFX Arrays **nicht** als Objekte gesehen und es stehen damit keine Eigenschaften oder Methoden zur Verfügung. In Java sind es Objekte.

Ebenso sind in JavaFX keine verschachtelten Arrays möglich.

Das Erzeugen von Arrays in JavaFX erfolgt über eine eigene Syntax. Die Werte eines Arrays können einfach, durch Kommata getrennt, bei einer Zuweisung in **eckigen** (nicht geschweiften wie in Java) Klammern notiert werden.

Beispiel:

```
var nums = [1, 2,3];
```

Listing 3.33: Anlegen eines Arrays

Diese Syntax kann auch auf zwei Anweisungen verteilt werden, was bei der verwandten Array-Erzeugung mit direkter Initialisierung in Java nicht geht.

```
var nums;
nums  = [1, 2,3];
```

Listing 3.34: Anlegen eines Arrays in zwei Schritten

Die Unterschiede in der Array-Erzeugung mit direkter Initialisierung in Java leiten dazu über, dass auch andere Syntaxkonstrukte aus Java in JavaFX nicht funktionieren. Etwa die Java-Form zum Deklarieren einer Referenzvariable auf ein Array oder das Anlegen eines Arrays, wie es in Java verwendet wird (oder zumindest eine daran maximal angepasste Version). Also wäre so etwas wie hier **falsch**:

```
var x[];
var y = {1,2,3};
var z[] = {1,2,3};
```

Listing 3.35: So etwas geht alles in JavaFX nicht

Allerdings können Sie in JavaFX zum Anlegen eines Arrays Folgendes machen (das ist dann weitgehend[97] analog wie in Java):

```
var y = new Integer[5];
```

Listing 3.36: Eine weitgehend der Java-Syntax entsprechende Form des Anlegens eines Arrays

Was Arrays in JavaFX noch massiv auszeichnet, ist, dass Sie hier auch den Variablenbezeichner ganz ohne Index verwenden können. Oder genauer – der Index lässt sich als **XPath**- respektive **XQuery**-Ausdruck verstehen.

Schauen wir uns zur Verdeutlichung ein vollständiges Beispiellisting an (*javafxapp2*):

```
01 import java.lang.*;
02
03 var nums;
04 nums  = [1, 2,3];
05
06 System.out.println(nums[0]);
07 System.out.println(nums);
08 System.out.println(nums[1]);
09 System.out.println(nums[2]);
```

Listing 3.37: Der Zugriff auf die einzelnen Werte in der Variablen nums

In dem Beispiel wird eine Variable `nums` mit mehreren Werten angelegt. Die Werte werden dann in der Konsole ausgegeben.

Die Ausgabe ist folgende:

1

1

2

3

Beachten Sie insbesondere Zeile 7!

97 Außer der Deklaration der Variablen mit `var`.

In JavaFX ist `[]` ein eigener **Operator** und dieser Operator `[]` steht für eine **Selektion**. Die darin notierte Lokalisierung ist eine Verwendung, die wie gesagt ähnlich in **XPath** beziehungsweise **XQuery** vorkommt.

> > > HINWEIS

*Unter **XML Path Language** oder kurz **XPath** versteht man eine vom W3C entwickelte Lokalisierungssprache, um Bestandteile in einem XML-Dokument ansprechen zu können. Ein XPath-Ausdruck versteht ein XML-Dokument als Baum. Dieser ist aus so genannten **Knoten** (**nodes**) und **Achsen** (**axis**) aufgebaut. Als Knoten des Baumes werden alle Bestandteile des XML-Dokuments (Elemente, Attribute, Kommentare, etc.) gesehen, die in einer Hierarchie einsortiert werden. Achsen in dem Baum werden in XPath wie der Stamm beziehungsweise die Äste des Baums verstanden und berücksichtigen auch die Reihenfolge der Elementdeklarationen (aus sequentieller Sicht).*

Wie in einem Verzeichnispfad können Sie unter XPath einen Knoten oder eine Achse ansprechen. Bei der Angabe wird immer der aktuelle Knoten in der Baumstruktur des XML-Dokuments als Basis verwendet. Dies ist vollkommen analog der Art und Weise, wie Sie Pfadangaben in einem Verzeichnisdienst machen. Auch dort stehen Sie in einem aktuellen Verzeichnis und geben von da den Weg zu einem anderen Ort (einem anderen Verzeichnis oder einer Datei) an.

*Unter XPath verwenden Sie zur Lokalisierung einen oder mehrere **Lokalisierungsschritte** (**Location Steps**), die gegebenenfalls wie unter Unix mit dem Token / getrennt werden. Wollen Sie die Wurzel des XML-Dokuments ansprechen, wird ein XPath-Ausdruck (wie unter Unix) direkt mit dem Token / begonnen. Ein Lokalisierungsschritt besteht immer aus der Angabe einer Achse und einem so genannten **Knotentest** (**node-test**), dem optional **Prädikate** (**predicates**) folgen können. Dabei lassen sich beliebig viele XPath-Ausdrücke mit dem Oder-Operator | verbinden und Achsen können gelegentlich mit zusammengesetzten Bezeichnern angesprochen werden, wenn dies einen XPath-Ausdruck verkürzt. Ebenso gibt es für Pfadangaben auf Achsen einige Abkürzungen, die der Notation in Verzeichnisangaben sehr ähnlich sind.*

*In Bezug auf einige Anwendungen in JavaFX ist nun vor allem eine auf XPath aufsetzende Sprache mit Namen **XQuery** (**XML Query**) von Bedeutung. Dies ist eine standardisierte Abfragesprache für XML-Datenbanken, die vom World Wide Web Consortium entwickelt wurde und dort eine ähnliche Funktion ausüben soll wie **SQL** (ursprünglich die Abkürzung für **Structured Query Language** – seit einigen Jahres steht es für **Standard Query Language**) für relationale Datenbanken. XQuery ist auch nicht auf XML-Daten beschränkt, sondern kann für verschiedene Abfragen eingesetzt werden – eben auch im Zusammenhang mit JavaFX, was wir unter anderem in diesem Abschnitt sehen werden.*

Aus dieser Xpath angelehnten Sicht ist der Ausdruck in den eckigen Klammern `[]` ein Boolescher Ausdruck[98]. Und dieser gibt ein neues Array zurück, das nur die Elemente enthält, die durch diesen Ausdruck in den eckigen Klammern beschrieben werden.

98 Diese Sichtweise unterscheidet sich gänzlich von der in Java.

Sie formulieren also in den eckigen Klammern einen Lokalisierungsausdruck und erhalten damit eine Teilmenge des Originaldatenfeldes.

> > > HINWEIS

JavaFX tritt mit dem Anspruch an, die Programmierung von Java-Applikation zu erleichtern. Jedoch habe ich in meinen XML-Kursen die Erfahrungen gemacht, dass sowohl XPath als auch XQuery ganz und gar nicht so einfach und verständlich sind, wie es immer propagiert wird. Im Gegenteil – für Einsteiger ist der Umgang mit solchen Lokalisierungsschritten nach meiner Erfahrung sogar extrem schwierig.

Die Motivation für die Einführung in JavaFX dürfte aber sein, dass es mittlerweile dennoch ein recht verbreitetes Wissen im XML-Umfeld gibt (auch bei Leuten, die nicht programmieren) und man diese Syntax alternativ zu einer klassischen Programmierung für den Zugriff auf Teile eines Arrays verwenden kann – immerhin soll sich JavaFX auch Nichtprogrammierern verständlich zeigen. Oder um es so zu formulieren – wenn Ihnen diese Syntax gefällt und Sie damit klarkommen, ist es wunderbar. Dann sollten Sie diese einsetzen. Wenn Sie sie aber nicht verstehen, können Sie sämtliche Lokalisierungen auf einem vollständigen Array auch mit der klassischen Programmierung über Bedingungen und Schleifen durchführen. Sie können diese Lokalisierung also als ein Angebot von JavaFX verstehen, das Sie auch ausschlagen können, um stattdessen mit klassischen Programmiertechniken zu arbeiten.

3.7.3 Abfragen auf Arrays

JavaFX stellt eine größere Anzahl an Lokalisierungsausdrücken zur Verfügung. Sie können zum Beispiel Vergleiche auf den Index formulieren.

Dabei deklarieren Sie in der Regel auf eine etwas ungewohnte Weise (mit einem |-Operator) eine Variable für das aktuelle Kontextobjekt, auf dem Sie eine Abfrage zur Ermittlung eines Teildatenfeldes formulieren.

Beispiel:

```
var nums = [1,2,3,4];
var numsGreaterThanTwo = nums[n|n > 2]; // liefert [3, 4]
```

Listing 3.38: Auswahl eines Teildatenfeldes

> > > HINWEIS

Wie auch in XPath kann man nach den offiziellen Dokumentationen in JavaFX bei einem Testausdruck mit dem Punktoperator auf das Kontextobjekt zugreifen. Schauen wir uns eine entsprechende Variante unseres Beispiels an:

```
var nums = [1,2,3,4];
var numsGreaterThanTwo = nums[. > 2]; // liefert [3, 4]
```

Listing 3.39: Eine Alternative zur Auswahl eines Teildatenfeldes

Interessanterweise macht diese Syntax, die auch in vielen offiziellen Beispielen zu JavaFX Verwendung findet, beim Übersetzen Probleme. Bei diversen Konstellationen kommt bei der Verwendung des Punktoperators eine Fehlermeldung der folgenden Art:

Error in file: ... incompatible types: expected Number, found Boolean in ...

Die zuerst genannte Schreibweise funktioniert aber meist einwandfrei, wenn man sie alternativ einsetzen kann. Hier zeigt sich wahrscheinlich auch, dass zum Zeitpunkt der Bucherstellung JavaFX noch in einer Betaversion vorliegt.

In JavaFX gibt es den Operator `indexof`. Dieser gibt die ordinale Position von einem Element innerhalb eines Arrays zurück[99].

In JavaFX wird das leere Array mit `[]` bezeichnet. Dies ist ein identischer Ausdruck zu dem Token `null`. Deshalb liefert ein Vergleich auch `true`.

3.7.4 Die Größe eines Arrays – sizeof

Die Größe von einem Array[100] kann man in Java ganz einfach bestimmen, da jedes Array ein Objekt ist. Dies ist wie gesagt in JavaFX nicht so. Zur Bestimmung der Größe von einem Array in JavaFX muss man einen Operator verwenden – den `sizeof`-Operator.

Beispiel:

```
var n = sizeof num;
```

Listing 3.40: die Größe von einem Array

Schauen wir uns ein vollständiges Beispiel mit einem leeren Array und `sizeof` an (*leeresarray*):

```
01 import java.lang.System;
02
03 System.out.println([] == null);
04 System.out.println(sizeof []);
05 System.out.println(sizeof null);
```

Listing 3.41: Das leere Array

Die Ausgabe ist Folgende:

true

0

0

99 Das entspricht exakt der Funktion `position()` in XPath.
100 Also die Anzahl der enthaltenen Elemente.

3.7.5 Eine Kurznotation für Arrays mit einer arithmetischen Serie

Es gibt in JavaFX eine **Kurznotation** für Arrays mit einer arithmetischen Serie (eine lineare Folge an Zahlen). Man notiert in die eckigen Klammern einfach zwei Punkte (..) zwischen den Start- und Endwert für die Zahlenfolge. Wenn Sie am Beginn des Wertebereichs zwei Zahlen angeben, die Sie mit einem Komma trennen, geben Sie eine **Schrittweite** der Werte an, mit der das Datenfeld gefüllt wird. Indirekt legen Sie damit natürlich dann auch die Anzahl der Werte in dem Datenfeld und damit die Größe von dem Datenfeld fest.

Hier sehen Sie zum Beispiel die Definition einer Funktion, die als Rückgabewert alle Zahlen in einem Datenfeld addiert, sowie eine zweite Funktion, die nur die geraden Zahlen addiert (*arrayarithmetischeserie*):

```
01 import java.lang.System;
02
03 // Addition aller Zahlen von 1 bis zum Wert des Übergabewertes
04 operation addZahlen(n) {
05   var x = 0;
06   for(i in [1..n]) {
07     x +=i;
08   }
09   return x;
10 }
11
12 // Addition aller geraden Zahlen von 2 bis zum Wert des Übergabewertes
13 operation addGeradeZahlen(n) {
14   var x = 0;
15   for(i in [2,4..n]) {
16     x +=i;
17   }
18   return x;
19 }
20
21 // Summe von 1 bis 5
22 System.out.println(addZahlen(5));
23 // Summe aller geraden Zahlen von 2 bis 10
24 System.out.println(addGeradeZahlen(10));
25 // Größe des Array von 1 bis 100 mit allen Zahlen
26 System.out.println(sizeof [1..100]);
27 // Größe des Array von 1 bis 100 mit allen Zahlen und der Schrittweite 4
28 System.out.println(sizeof [1,5..100]);
```

Listing 3.42: Datenfeldern mit Zahlenfolgen

Die Ausgabe ist Folgende:

15

30

100

25

3.7.6 Die Modifikation von Arrays – Elemente einfügen

Allgemein werden Werte von Arrays über die Zuweisung eines neuen Werts modifiziert. Arrays sind nun aber in Java bei weitem nicht so flexibel zu handhaben, wie man es möglicherweise annimmt. Java verwendet so genannte **semidynamische** Datenfelder, da die Größe eines Datenfelds zwar nicht bei der Deklaration festgelegt werden muss, sondern sich auch erst zur Laufzeit ergeben kann. Das ist der dynamische Anteil.

Das vorangestellte **semi** bedeutet, dass in Java ein Datenfeld jedoch nicht mehr in der Größe verändert werden kann, wenn es einmal in der Größe festgelegt wurde. Sie können also keine weiteren Elemente hinzufügen, noch können Sie vorhandene Elemente löschen[101].

Sollten Sie beispielsweise ein Datenfeld mit 100 Einträgen erzeugt haben und Sie benötigen 120 Elemente, dann müssen Sie ein neues Datenfeld erzeugen und die Werte der Elemente aus dem alten Datenfeld hineinkopieren.

Das ist absolut nicht schwierig, aber recht unbequem. Aus diesem Grund hat man sukzessive in Java Erweiterungen von normalen Datenfeldern eingeführt, die eine vollständige dynamische Veränderung gestatten. JavaFX verwendet nun Mechanismen, die einen dynamischen Umgang mit Datenfeldern gestatten[102].

Zusätzlich zu dem Zuweisungsoperator (=) zum Verändern eines Werts von einem Element gibt es in JavaFX Operatoren zur dynamischen Modifikation der Anzahl von Elementen in Arrays (**data modification operators**). Dies sind `insert` und `delete`. Sie werden identisch in der Syntax als auch Semantic eingesetzt wie diejenigen, die in der zukünftigen Version von XQuery Verwendung finden sollen.

101 Natürlich können Sie einen Wert auf `null` oder sonst einen leeren Wert setzen.

102 Wie das genau im Hintergrund mit Java gehandhabt wird, ist dabei vollkommen egal.

Einfügen von Elementen – die insert-Anweisung

Mit der insert-Anweisung fügen Sie Elemente zu einem bestimmten Datenfeld hinzu. Diese Anweisung kann verschiedene Formen annehmen:

```
insert Ausdruck1 [as first | as last] into Ausdruck2
insert Ausdruck1 before Ausdruck2
insert Ausdruck1 after Ausdruck2
```

Listing 3.43: Schemata für die unterschiedliche Verwendung von insert

Die Anweisung `insert` fügt die Elemente, die sich durch die Evaluierung von `Ausdruck1` ergeben, an einer bestimmten Stelle in das Datenfeld ein. Dabei muss `Ausdruck2` hinter dem Schlüsselwort `into` auf ein Attribut eines Objekts oder eine Variable verweisen.

> > > HINWEIS

Wenn `Ausdruck2` auf ein Attribut mit nur einem Wert verweist, hat der Operator `insert` den gleichen Effekt wie der Zuweisungsoperator.

- Wenn der Token `as first` angegeben wird, erfolgt die Einfügung des neuen Elements beziehungsweise der Elemente vor dem ersten Element der Liste, die mit `Ausdruck2` spezifiziert wird.
- Wenn stattdessen der Token `as last` angegeben wird, erfolgt vollkommen analog das Einfügen hinter dem letzten Element der mit `Ausdruck2` spezifizierten Liste.
- Wenn weder `as first` noch `as last` explizit angegeben werden, gilt als Vorgabe `as last`.

Beispiel (*arrayinsert*):

```
01 import java.lang.System;
02
03 // Operation zur Ausgabe wichtiger Eckdaten des Arrays
04 operation auswertungArray(x){
05   // Größe des Arrays
06   System.out.println("Groesse des Arrays: {sizeof x}");
07   // interner Zähler für die folgende Schleife
08   var counter = 0;
09   // Ausgabe aller Elemente in dem Array
10   for(i in x) {
11     System.out.println("Wert von Index {counter++}: {i}");
12   }
13   // Trennlinie
14   System.out.println("_____________________________");
15 }
16
```

```
17 // Anlegen eines Arrays
18 var x = [1,2,3];
19 auswertungArray(x);
20
21 // Einfügen eines Wertes - Defaulteinstellung
22 insert 12 into x;
23 System.out.println("Anweisung: insert 12 into x ");
24 auswertungArray(x);
25
26 // Einfügen eines Wertes am Anfang
27 insert 10 as first into x;
28 System.out.println(
29   "Anweisung: insert 10 as first into x");
30 auswertungArray(x);
31
32 // Einfügen zweier Werte (Array) am Ende
33 insert [99,100] as last into x;
34 System.out.println(
35   "Anweisung: insert [99,100] as last into x");
36 auswertungArray(x);
37
38 // Einfügen einer arithmetischen Serie am Anfang
39 insert [40,44 .. 55] as first into x;
40 System.out.println(
41   "Anweisung: insert [40,44 .. 55] as first into x");
42 auswertungArray(x);
```

Listing 3.44: Einfügen von Elementen

In dem Beispiel finden Sie von der Zeile 4 bis zu Zeile 15 eine Operation (`auswertungArray()`), die die wichtigsten Eckdaten des Arrays ausgibt. Wir rufen diese Operation nach jeder Veränderung des Arrays auf um zu kontrollieren, wie sich die Größe des Arrays geändert hat und welche Elemente nun darin enthalten sind. Die Operation bekommt als Übergabewert eine Referenz auf das Array übergeben.

Die Variable `x` wird in Zeile 18 deklariert und mit einem Defaultwert (ein Array mit drei Einträgen) belegt.

In Zeile 22 sehen Sie das Einfügen eines Wertes mit der Defaulteinstellung. Damit wird das neue Element am Ende des bestehenden Arrays eingefügt (`insert 12 into x;`).

In Zeile 27 sehen Sie dagegen das Einfügen eines Wertes am Anfang des bestehenden Arrays mit der Anweisung `insert 10 as first into x;`.

Die Zeile 33 zeigt das Einfügen zweier Werte (also eines Arrays), die explizit an das Ende des bestehenden Arrays angefügt werden (`insert [99,100] as last into x;`).

Die Aktion in Zeile 39 zeigt Ihnen das Einfügen einer arithmetischen Serie am Anfang eines bestehenden Arrays (`insert [40,44 .. 55] as first into x;`).

Das ist die Ausgabe des Beispiels:

Groesse des Arrays: 3

Wert von Index 0: 1

Wert von Index 1: 2

Wert von Index 2: 3

Anweisung: insert 12 into x

Groesse des Arrays: 4

Wert von Index 0: 1

Wert von Index 1: 2

Wert von Index 2: 3

Wert von Index 3: 12

Anweisung: insert 10 as first into x

Groesse des Arrays: 5

Wert von Index 0: 10

Wert von Index 1: 1

Wert von Index 2: 2

Wert von Index 3: 3

Wert von Index 4: 12

Anweisung: insert [99,100] as last into x

Groesse des Arrays: 7

Wert von Index 0: 10

Wert von Index 1: 1

Wert von Index 2: 2

Wert von Index 3: 3

Wert von Index 4: 12

Wert von Index 5: 99

Wert von Index 6: 100

Anweisung: insert [40,44 .. 55] as first into x

Groesse des Arrays: 11

Wert von Index 0: 40

Wert von Index 1: 44

Wert von Index 2: 48

Wert von Index 3: 52

Wert von Index 4: 10

Wert von Index 5: 1

Wert von Index 6: 2

Wert von Index 7: 3

Wert von Index 8: 12

Wert von Index 9: 99

Wert von Index 10: 100

Die Angaben before und after

Wie Sie eben gesehen haben, können Sie Elemente am Ende oder am Anfang eines bestehenden Arrays einfügen. Mit JavaFX können Sie aber auch an einer beliebigen Stelle im Array neue Elemente hinzufügen. Dazu dienen die Angaben `before` und `after`. `Ausdruck2` muss ein Selektionsausdruck über ein Attribut oder eine Variable sein. Wenn der Token `before` angegeben wird, erfolgt die Einfügung vor dem spezifizierten Element. Entsprechend erfolgt die Einfügung bei `after` danach.

Beispiel (*arrayinsertbevorafter* – die Operation ist identisch zu dem vorherigen Beispiel und wird hier aus Platzgründen nicht mehr abgedruckt):

```
01 import java.lang.System;
02
03 // Operation zur Ausgabe wichtiger Eckdaten des Arrays
04 operation auswertungArray(x){
...
15 }
16
17 // Anlegen eines Arrays
18 var x = [1,2,3];
19 auswertungArray(x);
20
21 // Einfügen eines Wertes nach dem Element mit dem Wert 1
```

```
22 insert 10 after x[n|n == 1];
23 System.out.println("Anweisung: insert 10 after x[n|n == 10]");
24 auswertungArray(x);
25
26 // Einfügen eines Wertes an der 2. Stelle des bestehenden Arrays
27 insert 12 before x[1];
28 System.out.println("Anweisung: insert 12 before x[1]");
29 auswertungArray(x);
30
31 // Einfügen eines Wertes nach dem Element mit dem Wert 2
32 insert 13 after x[n|n == 2];
33 System.out.println("Anweisung: insert 13 after x[n|n == 2]");
34 auswertungArray(x);
```

Listing 3.45: Das gezielte Einfügen an einer beliebigen Stelle im bestehenden Array

In der Zeile 22 sehen Sie das Einfügen eines Wertes nach dem Element mit dem Wert 1 (`insert 10 after x[n|n == 1];`). Das bedeutet, dass der Wert 10 in dem vorher vorhandenen Array zwischen den Werten 1 und 2 eingefügt wird. Er steht damit in dem neuen Array an der Position 2 (das ist der Index 1).

In Zeile 27 sehen Sie das explizite Einfügen eines Wertes an einer bestimmten Stelle im bisherigen Array, die über einen Index spezifiziert wird. Das ist in diesem Fall die zweite Stelle des bestehenden Arrays (`insert 12 before x[1];`).

In Zeile 32 erfolgt erneut das Einfügen eines Wertes nach einem Element, das über einen Wert spezifiziert wird (`insert 13 after x[n|n == 2];`).

Die Ausgabe des Beispiels ergibt das:

Groesse des Arrays: 3

Wert von Index 0: 1

Wert von Index 1: 2

Wert von Index 2: 3

Anweisung: insert 10 after x[n | n == 10]

Groesse des Arrays: 4

Wert von Index 0: 1

Wert von Index 1: 10

Wert von Index 2: 2

Wert von Index 3: 3

Anweisung: insert 12 before x[1]

Groesse des Arrays: 5

Wert von Index 0: 1

Wert von Index 1: 12

Wert von Index 2: 10

Wert von Index 3: 2

Wert von Index 4: 3

Anweisung: insert 13 after x[n | n == 2]

Groesse des Arrays: 6

Wert von Index 0: 1

Wert von Index 1: 12

Wert von Index 2: 10

Wert von Index 3: 2

Wert von Index 4: 13

Wert von Index 5: 3

3.7.7 Das Löschen von Elementen in einem Array – die delete-Anweisung

Die Anweisung `delete` löscht Elemente aus einem vorhandenen Datenfeld. Dabei kann diese Anweisung folgende Formen annehmen:

```
delete Variable
delete Ausdruck.Attribut
delete Variable[Prädikat]
delete Ausdruck.Attribut[Prädikat]
```

Listing 3.46: Schemata für alternative Formen für die delete-Anweisung

- Die ersten zwei Varianten entfernen alle Elemente aus einer Variablen oder einem Attribut. Dies ist äquivalent zu der Zuweisung von `[]` oder `null` zu der Variablen oder dem Attribut.
- Die letzten beiden Varianten entfernen nur die Elemente, die mit dem Prädikat (einem Testausdruck) übereinstimmen.

Beispiel (*arraydelete* – die Operation ist identisch zu dem vorherigen Beispiel und wird nicht mehr abgedruckt):

```
01 import java.lang.System;
02
03 // Operation zur Ausgabe wichtiger Eckdaten des Arrays
04 operation auswertungArray(x){
...
15 }
16
17 // Anlegen eines Arrays
18 var x = [1, 2, 3, 4, 5, 6, 7, 8 , 1, 3, 5, 7, 9, 11];
19 auswertungArray(x);
20
21 // Löschen des Eintrags mit dem Wert 6
22 System.out.println("Anweisung: delete x[n|n == 6]");
23 delete x[n|n == 6];
24 auswertungArray(x);
25
26 // Löschen aller Einträge mit dem Wert >= 3
27 delete x[n|n >= 3];
28 System.out.println("Anweisung: delete x[n|n >= 3]");
29 auswertungArray(x);
30
31 // Löschen aller Einträge
32 System.out.println("Anweisung: delete x");
33 delete x;
34 auswertungArray(x);
```

Listing 3.47: Löschen von Array-Elementen

In dem Beispiel arbeiten wir dieses Mal mit einem etwas größeren Array. Wir wollen ja einige Elemente löschen. Es sollte Ihnen auch auffallen, dass sich einige Werte in dem Array wiederholen.

In Zeile 23 erfolgt das gezielte Löschen des Eintrags mit dem Wert 6 (`delete x[n|n == 6];`).

In Zeile 27 löschen wir alle Einträge mit dem Wert >= 3 (`delete x[n|n >= 3];`). Da der Wert 1 im Original-Array zwei Mal vorkommt, bleibt ein Array mit drei Einträgen übrig.

Die Zeile 33 löscht alle Einträge in dem verbleibenden Array (`delete x;`).

Das ist die Ausgabe des Beispiels:

Groesse des Arrays: 14

Wert von Index 0: 1

Wert von Index 1: 2

Wert von Index 2: 3

Wert von Index 3: 4

Wert von Index 4: 5

Wert von Index 5: 6

Wert von Index 6: 7

Wert von Index 7: 8

Wert von Index 8: 1

Wert von Index 9: 3

Wert von Index 10: 5

Wert von Index 11: 7

Wert von Index 12: 9

Wert von Index 13: 11

Anweisung: delete x[n | n == 6]

Groesse des Arrays: 13

Wert von Index 0: 1

Wert von Index 1: 2

Wert von Index 2: 3

Wert von Index 3: 4

Wert von Index 4: 5

Wert von Index 5: 7

Wert von Index 6: 8

Wert von Index 7: 1

Wert von Index 8: 3

Wert von Index 9: 5

Wert von Index 10: 7

Wert von Index 11: 9

Wert von Index 12: 11

Anweisung: delete x[n | n >= 3]

Groesse des Arrays: 3

Wert von Index 0: 1

Wert von Index 1: 2

Wert von Index 2: 1

Anweisung: delete x

Groesse des Arrays: 0

3.7.8 Auswählen von Elementen – die select- und die foreach-Anweisung

JavaFX unterstützt über die Anweisung `select` sowie die schon an anderer Stelle behandelte Anweisung `foreach` die Auswahl von Elementen. Das Ergebnis wird eine Liste mit Treffern sein, die eine Teilmenge des bisherigen Arrays darstellen. Dabei kann man optional mit der Erweitung `where` einen **Filter** (einen Testausdruck) spezifizieren. Die beiden Anweisungen entsprechen in hohem Maß der `select`-Anweisung in SQL.

Hier ist ein Schema für die Verwendung:

```
select Ausdruck from Array [where Bedingung]
```

Listing 3.48: Ein Schema für die Anwendung der select-Anweisung

Schauen wir uns ein Beispiel an (*selectanweisung*):

```
01 import java.lang.System;
02
03 // Die Quadrate der Zahlen 1 bis 10
04 var zahlen = select n*n from n in [1..10];
05 for(i in zahlen) {
06   System.out.println(i);
07 }
08
09 // Alle geraden Zahlen zwischen 1 und 30 (inklusive)
10 zahlen = select j from j in [1..30] where j % 2 == 0;
11 for(i in zahlen) {
12   System.out.println(i);
13 }
```

Listing 3.49: Die Anwendung von select

Die Anweisung in Zeile 4 gibt die Quadrate der Zahlen 1 bis 10 als Ergebnis. Dabei ist `n` als eine **lokale Variable** des Ausdrucks zu sehen.

Die Anweisung in Zeile 9 liefert alle geraden Zahlen zwischen 1 und 30 (inklusive). Dabei kommt ein Filter zum Einsatz, der mittels des Schlüsselworts `where` eingeleitet wird.

Hier ist noch ein umfangreicheres Beispiel mit dem Einsatz von `select` und `foreach` (*selectanweisung1*):

```
01 import java.lang.System;
02
03 class Album {
04     attribute title: String;
05     attribute artist: String;
06     attribute tracks: String*;
07 }
08 var albums =
09 [Album {
10      title: "A Hard Day's Night"
11      artist: "The Beatles"
12      tracks:
13      ["A Hard Day's Night",
14       "I Should Have Known Better",
15       "If I Fell",
16       "I'm Happy Just To Dance With You",
17       "And I Love Her",
18       "Tell Me Why",
19       "Can't Buy Me Love",
20       "Any Time At All",
21       "I'll Cry Instead",
22       "Things We Said Today",
23       "When I Get Home",
24       "You Can't Do That"]
25 },
26 Album {
27       title: "Circle Of Love"
28       artist: "Steve Miller Band"
29       tracks:
30       ["Heart Like A Wheel",
31         "Get On Home",
32         "Baby Wanna Dance",
33         "Circle Of Love",
34         "Macho City"]
35 }];
36
37 var counter = 0;
38 // Abfrage aller Titelnummern, die mit dem Titel des jeweiligen
39 // Albums übereinstimmen
```

```
40
41 // Anwendung des select-Operators:
42 System.out.println("Anwendung des select-Operators:");
43 var titleTracks =
44     select indexof track + 1 from album in albums,
45                track in album.tracks
46                    where track == album.title;
47 for(i in titleTracks) {
48   System.out.println(
49     "Im {++counter}. Album stimmt der Titel {i} mit dem Namen des Albums
        ueberein.");
50   }
51
52 // Alternative über den Operator foreach
53 counter = 0;
54 System.out.println("Anwendung des foreach-Operators:");
55 titleTracks =
56     foreach (album in albums,
57                track in album.tracks
58                    where track == album.title)
59         indexof track + 1;
60 for(i in titleTracks) {
61     System.out.println(
62   "Im {++counter}. Album stimmt der Titel {i} mit dem Namen des Albums
      ueberein.");
63   }
```

Listing 3.50: Die Anwendung von select und foreach

Das Beispiel arbeitet mit der deklarativen Erzeugung von Objekten, die wir im nächsten Kapitel genauer besprechen werden. Bis zur Zeile 35 sollten Sie den Quellcode deshalb erst einmal so hinnehmen. Die beiden erzeugten Objekte repräsentieren Musikalben mit einer gewissen Anzahl an enthaltenen Stücken.

In Zeile 37 definieren wir eine Zählvariable (`var counter = 0;`), die einfach die nachfolgenden Ausgaben etwas verständlicher aufbereiten soll.

In der Zeile 43 bis 46 sehen Sie eine komplexere select-Anweisung (`var titleTracks = select indexof track + 1 from album in albums, track in album.tracks where track == album.title;`). Diese dient zur Abfrage aller Titelnummern, die mit dem Titel des jeweiligen Albums übereinstimmen. Das Ergebnis der Abfrage sind die jeweiligen Titelnummern.

In der folgenden Schleife wird das Ergebnis optisch etwas aussagekräftiger aufbereitet.

In den Zeilen 56 bis 59 sehen Sie eine vollkommen äquivalente Abfrage. Sie wird nur über den Operator `foreach` ausgeführt.

Das ist die Ausgabe des Beispiels:

Anwendung des select-Operators:

Im 1. Album stimmt der Titel 1 mit dem Namen des Albums ueberein.

Im 2. Album stimmt der Titel 4 mit dem Namen des Albums ueberein.

Anwendung des foreach-Operators:

Im 1. Album stimmt der Titel 1 mit dem Namen des Albums ueberein.

Im 2. Album stimmt der Titel 4 mit dem Namen des Albums ueberein.

3.8 Bezeichner in französischen Anführungszeichen

In JavaFX wird jede Sequenz an Zeichen, die in so genannten **französischen Anführungszeichen** notiert sind (<<>>), als Bezeichner verstanden. Dies erlaubt es zum Beispiel JavaFX-Schlüsselwörter als Bezeichner für Klassen, Variablen, Funktionen oder Attribute zu verwenden. Das bedeutet, so etwas wäre theoretisch möglich:

```
var <<while>> = 100;
```

Listing 3.51: Ein Bezeichner für eine Variable, der mit einem Schlüsselwort übereinstimmt

Nun gehört es zu den elementarsten Regeln guten Programmierstils, keine Bezeichner zu verwenden, die auch nur ansatzweise Schlüsselwörtern ähneln[103]. Solch eine missverständliche Verwendung erschwert die Wartbarkeit gewaltig.

Aber es gibt dennoch tatsächlich eine sinnvolle Anwendung für diese Möglichkeit. Sie können damit **Java**-Methoden in JavaFX aufrufen, deren Namen wie Schlüsselwörter in **JavaFX** lauten.

Und solche Fälle gibt es in der Tat.

Hier wäre ein Beispiel:

```
import javax.swing.JTextArea;
var textArea = new JTextArea();
textArea.<<insert>>("Hello", 0);
```

Listing 3.52: Aufruf einer Java-Methode, die einem JavaFX-Schlüsselwort entspricht

103 So etwas kann man auf Grund der Relevanz von Groß- und Kleinschreibung in verschiedenen Sprachen machen, indem man zum Beispiel eine Variable `IF` nennt, wenn das Schlüsselwort `if` ist.