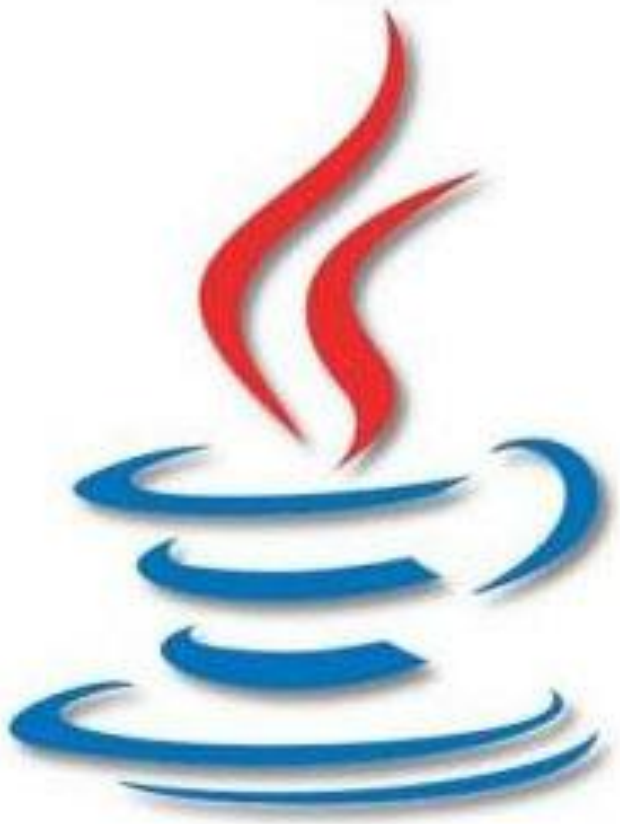
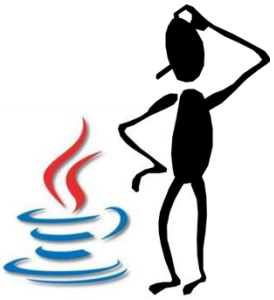


Inhalt

1 Einleitung

Anwendungen entwickeln ist schwer - zumindest wenn man es nicht kann. Um diesen Umstand zu ändern habe ich dieses Dokument geschrieben. Er soll all denen helfen, die sich einfach nicht an das Thema "Programmieren" heranwagen und sich sicher sind, dass Mutter Natur bei der Vergabe der Gene das "Programmiergen" bei ihnen vergessen hat. Ich kann euch jedoch beruhigen - ein solches Gen existiert nicht, bzw. konnte bis dato nicht nachgewiesen werden. Sollte von euch jemand das Gegenteil beweisen können, dann hat er ohnehin andere Fähigkeiten, die vielleicht auch ganz gut bezahlt werden und muss sich ums Programmieren keine Gedanken machen. Aber bis dahin kümmern wir uns erst mal um die Programmiererei.

Ich gehe bei diesem Kurs davon aus, dass ihr Java und Eclipse bereits installiert habt und wisst, wie man die Programme startet. Wenn nicht, dann empfehle ich euch meine kurze Anleitung
d://www.bs7-augsburg.de/aicher/files_codeconcert/10AnPr/ANPR_02_InstallationJAVA_Eclipse.pdf
Lest sie euch bitte durch (oder geht auf das YouTube Video
http://www.youtube.com/watch?v=4VpR-CXFiQo
und installiert die Programme auf euren Rechner. Ich warte solange hier - versprochen!

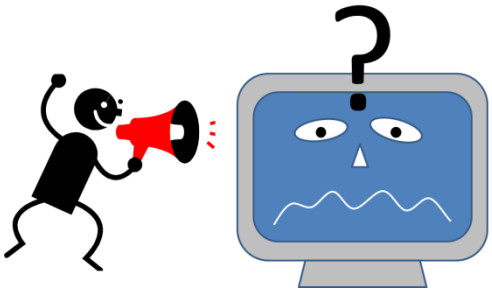


So - läuft alles? Wenn ja, dann kann's ja losgehen. Wenn nein, dann sucht euch bitte den nächsten Computerfreak der bei euch rumspringt und lasst euch dabei helfen die Umgebung aufzusetzen.

2 Das erste Programm

Wir wissen ja inzwischen aus leidiger Erfahrung, dass so ein Computer ein ziemlicher Dickkopf ist und eigentlich nie das macht, was man ihm sagt. Aber auch hier kann ich Entwarnung geben, er macht nämlich genau das, was man ihm sagt. Wir sind es nur nicht gewohnt, dass jemand genau das macht - also wirklich buchstäblich genau das macht, was wir sagen. Jaja, das ist leider so, keiner hört so wirklich auf uns oder? Wobei es ist eigentlich auch in Ordnung so... Wenn wir jedem haarklein alles erklären müssten, würden wir es im Regelfall gleich selbst tun - unsere Mitmenschen denken aber gottseidank mit und können auch mit ungenauen Anweisungen arbeiten. Der Computer aber nicht - der denkt keinen Schritt weiter! Und genau das müssen wir uns beim Programmieren immer vor Augen führen - der Computer ist zwar zuverlässig, aber strohdoof!

Dem nicht genug - der Computer versteht uns auch nicht wirklich. Er spricht keine "normalen" Sprachen - Deutsch schon zweimal nicht. Er kennt nur Programmiersprachen wie Java. Eigentlich kennt er nicht mal die - die Texte in Java (unsere Sourcen) müssen auch erst mal für ihn übersetzt werden. Man nennt dies "Compillieren". Nachdem er das Übersetzen vom Sourcecode mittels eines Compilers in eine ausführbare Datei allerdings selbst übernimmt, wollen wir da mal nicht so sein. Die Konsequenz aus dem Ganzen ist aber nun, dass wir eine neue Sprache lernen müssen. Hier gibt es verschiedene Varianten (C, C++, C#, Javascript, PHP usw.) - im Rahmen unserer Schule haben wir uns für die Sprache "Java" entschieden. Aber keine Angst; wenn man Java verstanden hat, dann wird man auch schnell den Zugang zu anderen Programmiersprachen finden.

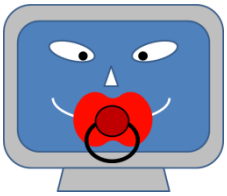


Wir halten also fest:

- Ein Computer versteht nur Anweisungen, die in einer für ihn ausgerichteten Sprache geschrieben wurden.
- Die "Sprache", in der wir dem Computer sagen, was er tun muss nennen wir "Programmiersprache". Hier gibt es verschiedene Programmiersprachen - wir werden für diesen Kurs "Java" verwenden.
- Nachdem die Dateien mit dem Programm in den entsprechenden Sprachen (auch Sourcedateien genannt) auch nur ein Zwischenschritt sind, müssen diese übersetzt werden, damit der Computer damit arbeiten kann. Diesen Vorgang nennt man Compillieren.
- Grundsätzlich führt der Computer die Anweisungen in unseren Programmen aus - ohne sich darüber Gedanken zu machen, ob sie sinnvoll sind oder nicht. Der Computer denkt nicht!

Zusatzinfo:
Ein Computer arbeitet digital. Er speichert also ausschließlich Nullen und Einsen ab. Mit Hilfe des Binärsystems lassen sich dadurch somit Zahlen erzeugen, die er im Endeffekt speichert. Im Umkehrschluss bedeutet dies aber, dass er ausschließlich Zahlen speichern und lesen kann. Jeder Text, jedes Musikstück und jede Grafik, die im Rechner residiert ist nichts anderes als eine strukturierte Ansammlung von Zahlen. Ein Programm ist somit auch nichts anderes als eine Ansammlung von Zahlen. Der Rechenkern eines Prozessors versteht somit auch nur Zahlen. Ein nativ compillierter Code (also Code, der direkt für die Nutzung auf einem Prozessor compilliert wurde) ist also eine sequenz von Zahlen, bei denen jede Zahl für eine Aktion im Prozessor (plus Daten) steht. Man spricht hier von Maschinencode - also Code, der direkt vom Prozessor ausgeführt werden kann. Bei Java kommt nun noch dazu, dass der compillierte Code nicht direkt auf dem Rechner läuft, sondern auf einem virtuellen Rechner, der sogenannten JVM (Java Virtual Machine).

Wie bei allen anderen Programmierkursen habe auch ich die heilige Verpflichtung, zuerst das "Hello World" Programm zu lehren. Ich weiß zwar nicht was passiert, wenn ich ein anderes Programm am Anfang eines Kurses setze, aber ich muss wohl davon ausgehen, dass mich sieben Jahre Pech verfolgen - insofern bleibe ich bei "Hello World".



Unter dem "Hello World" Programm versteht man eigentlich das einfachste denkbare Programm, welches lediglich einen kleinen Text auf die Konsole ausgibt - nämlich "Hello World". Wozu soll das nun gut sein? Im Prinzip erreicht man durch ein solches Programm, dass man die grundlegenden notwendigen Schritte beim Programmieren verstanden hat - eine Entwicklungsumgebung starten (das kann im Zweifelsfall auch Notepad.exe und der Javacompiler sein), das Grundgerüst für ein Programm erstellen und eine Zeile einfügen, welche etwas vorhersehbares macht (nämlich einen Text ausgeben). Weiterhin prüfen wir, ob wir unser Programm auch starten und somit testen können. Es geht also ausschließlich darum, die Nutzung unserer Umgebung initial zu üben. Dann fangen wir mal an - in Eclipse machen wir uns das Leben möglichst einfach, und erzeugen in einem Javaprojekt eine Javaklasse.

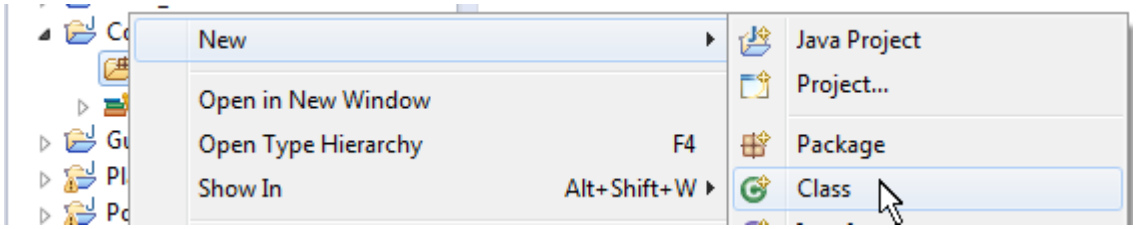
Bevor wir das tun, nochmal ein kurzer Exkurs zum Thema Projekt und Klasse:

Projekt:
Ein Projekt ist im Rahmen einer Programmierumgebung eine Ansammlung von zusammengehörigen Files, welche eine Funktion (oder auch Funktionen) ausführen - und zwar genau die Funktion(en), welche in der Projektdefinition als notwendig definiert wurden. Im einfachsten Fall ist ein Projekt nur ein einziges File, in dem unser Code steht, also für uns unser "HelloWorld" Programm. Bei komplizierteren Projekten finden wir im Regelfall eine Flut von Dateien - Sourcen, Bibliotheken, Grafiken usw. usf. Ziel einer Gruppierung in Projekte ist es, am Ende des Entwicklungsprozesses alle relevanten Dateien für ein Endprodukt zusammengehörig abgespeichert zu haben. Da wir am Anfang nur sehr einfache Programme schreiben, können wir unsere Projekte eher allgemein halten und bspw. alle Programme eines bestimmten Lernabschnittes zusammenfügen.

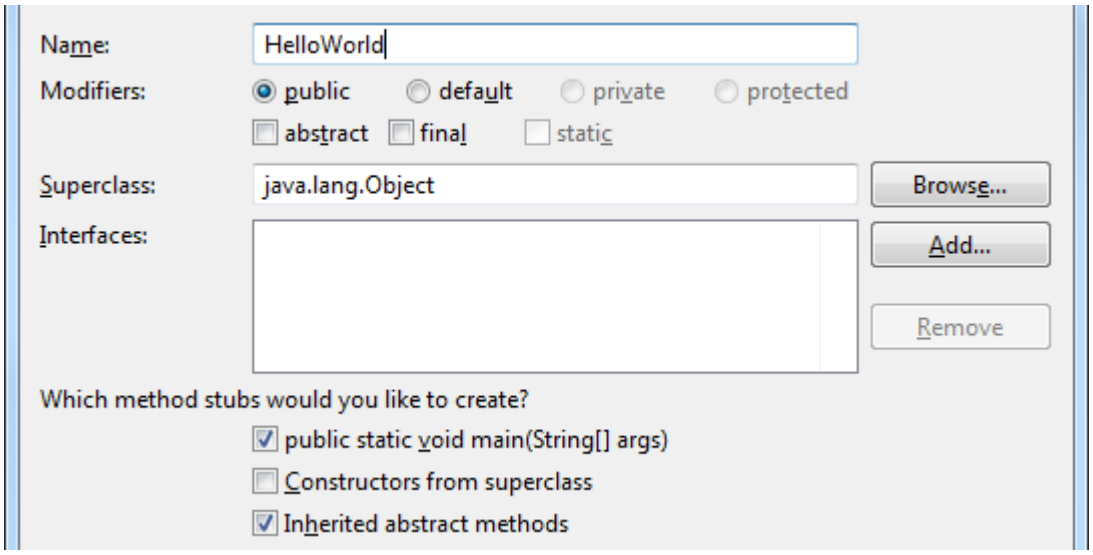
Klasse:

Java ist (wie viele anderen Programmiersprachen auch) objektorientiert. Das heißt, dass die einzelnen Programmelemente keine reinen Funktionen sind, sondern als mehr oder weniger selbstständige Objekte angesehen werden, die gewisse Eigenschaften und Funktionalitäten aufweisen. Ein Objekt kann somit Daten beinhalten und auf Basis dieser Daten gewisse Funktionen ausführen. Eine Klasse wiederum ist nichts anderes als ein Bauplan eines solchen Objektes. Übersetzt heißt dies, dass die Klasse das Programm ist, welches wir schreiben und das Objekt wiederum ein gewisser Speicherbereich während der Laufzeit ist, in dem die Daten und die Funktionalitäten residieren, welche laut unserem Code existieren müssen. Da wir am Anfang uns nicht um das Thema "Objektorientierung" kümmern werden gilt für uns erst mal nur die Devise - unser Programm ist die Klasse und wir starten dies über unsere Entwicklungsumgebung.

Wir erstellen also ein neues Javaprojekt in Eclipse, wie im Dokument
http://www.bs7-augsburg.de/aicher/files_codeconcert/10AnPr/ANPR_02_InstallationJAVA_Eclipse.pdf
geschrieben. Anschließend wird eine neue Javaklasse in diesem Projekt erstellt:



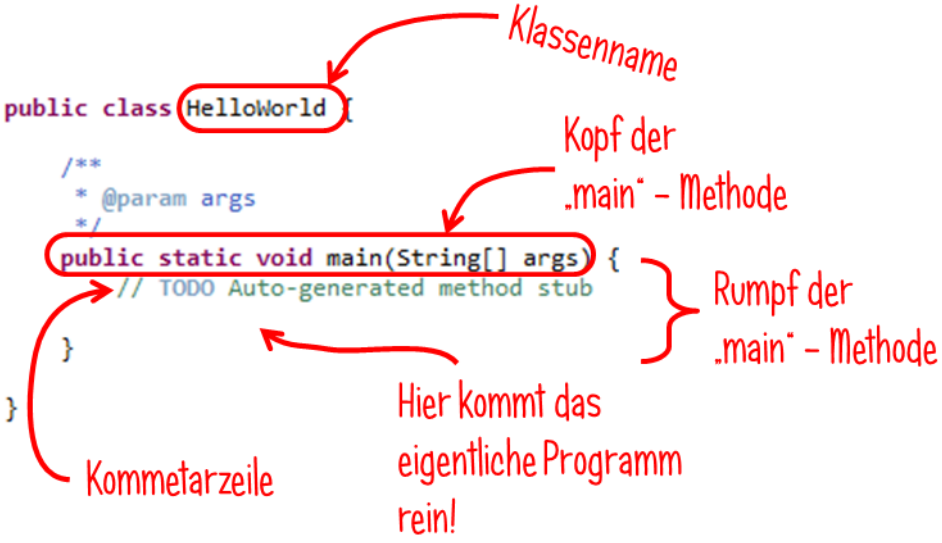
Diese Klasse taufen wir „HelloWorld“:



Bitte achtet auch darauf, dass der Haken bei „public static void main(String[] args)“ gesetzt ist, damit Eclipse einen kompletten Rumpf für eine eigenständige Klasse mitsamt Mainmethode erstellt.

Sobald wir auf „Finish“ klicken, erstellt Eclipse für uns ein Grundgerüst einer Klasse – für uns also eines eigenen Programms.

Sehen wir uns den Code, den Eclipse für uns erstellt hat mal etwas genauer an:



Wie wir sehen, taucht unser Klassenname „HelloWorld" im Code bereits auf. Vor dem Klassennamen stehen aber noch zwei „merkwürdige" Java Schlüsselwörter – „public" und „class". Nun – was class bedeuten soll können wir uns ja denken: es ist die Kennzeichnung, dass wir hier eine Klasse programmieren. Das Wort „public" soll heißen, dass die Klasse von jeder anderen Klasse genutzt werden kann. Dies wird interessant, wenn wir uns näher mit der Objektorientierung befassen werden.

Nach der Klassendefinition folgt die sogenannte „main" – Methode. Was hier passiert ist, dass wenn wir eine Klasse als Programm starten wollen, der Computer wissen muss wo er denn mit der Abarbeitung des Codes beginnen soll. Dies ist immer die „main" – Methode. Sie ist also der Einstiegspunkt der JVM (Java Virtual Machine). Auch hier finden wir wieder Schlüsselwörter. „Public" kennen wir bereits. „static" bedeutet, dass die Klasse „alleine" im Rechner ausgeführt werden kann und nicht als Objekt laufen muss. Dies werden wir auch später nochmals detaillierter unter die Lupe nehmen. Das Schlüsselwort „void" besagt, dass die „main" – Methode keinen Wert an den aufrufenden zurückgibt.
Was wir ebenfalls noch sehen können ist, dass in den Klammern nach dem Wort „main" noch etwas angegeben ist – die sogenannten „Parameter". Dies ist ein Array (also eine Liste von Werten), welche eventuelle Parameter beim Aufruf der Klasse beinhalten. Da wir aber vorerst die Klasse ohne Argumente aufrufen, müssen wir diese Parameter nicht weiter beachten.

Was nun folgt, ist eine Kommentarzeile, in der Eclipse uns darauf aufmerksam machen will, dass der Code automatisch generiert wurde und wir an dieser Stelle besser einen sinnvollen Code eintragen sollten – für unser Programm also der Befehl für die Ausgabe auf der Konsole.

Zusatzinfo:
Eine Kommentar bei der Programmierung ist ein Bereich, der von der ausführenden Instanz (bei Java also der JVM) nicht beachtet wird. Wir können in einer Kommentarzeile also beliebige Dinge hineinschreiben – im Wesentlichen um uns selbst ein paar Gedankenstützen einzutragen, was wir uns beim Codieren eigentlich gedacht haben. Es gibt zwei verschiedene Kommentarvarianten in Java. Der Zeilenkommentar, dies sind zwei Slashzeichen. Alles rechts von diesen beiden Zeichen wird somit ignoriert:

```
Hier wird der Code noch interpretiert; // hier wird er nicht mehr
// interpretiert.
```

Die zweite Alternative ist der Kommentarbereich. Dieser startet mit einem Slash, gefolgt von einem Stern. Danach wird nichts mehr interpretiert. Erst wenn wieder ein Stern gefolgt von einem Slash kommt, fängt der Interpreter wieder an den Code „ernst" zu nehmen:

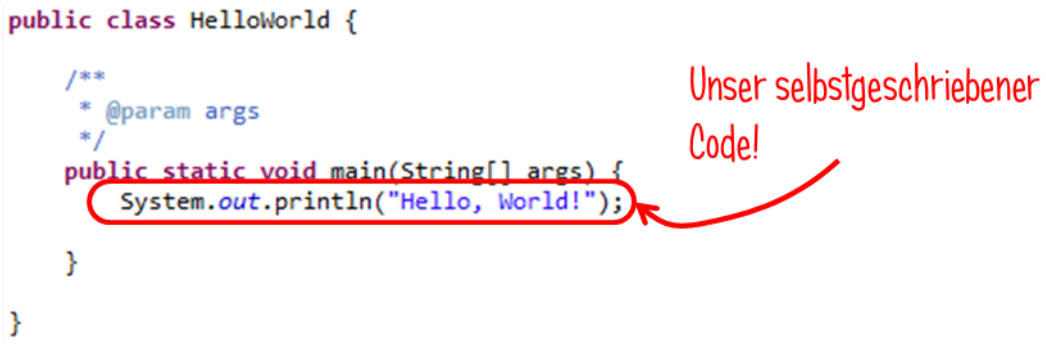
```
Hier wird der Code noch interpretiert /* hier wird der Code nicht mehr
interpretiert */ ab hier wird er wieder interpretiert.
```

Was uns weiterhin noch auffällt, sind die geschweiften Klammern. Diese dienen dazu, zusammengehörige Bereiche zu markieren. Alles, was zwischen einer sich öffnenden und sich schließenden Klammer steht, ist für den Interpreter eine „Ebene“. Das heißt, dass in unserem Beispiel alles was zwischen der öffnenden Klammer auf Höhe der „main“ – Methode und nach dem Kommentar einen Bereich markiert, der zur „main“ – Methode gehört (der „Rumpf“). Diese Art der Gruppierung werden wir bei allen Java Elementen sehen, die einen Rumpf mit mehr als einer Zeile aufweisen.

Nun – da wir uns um den automatisch generierten Code nun gekümmert haben, sollten wir uns um die eigentliche Funktion unseres kleinen Programms kümmern. Wir fügen in unsere Klasse nun die folgende Zeile ein – beachtet bitte auch das Semikolon am Ende (es schließt den Befehl ab):

```
System.out.println("Hello, World!");
```

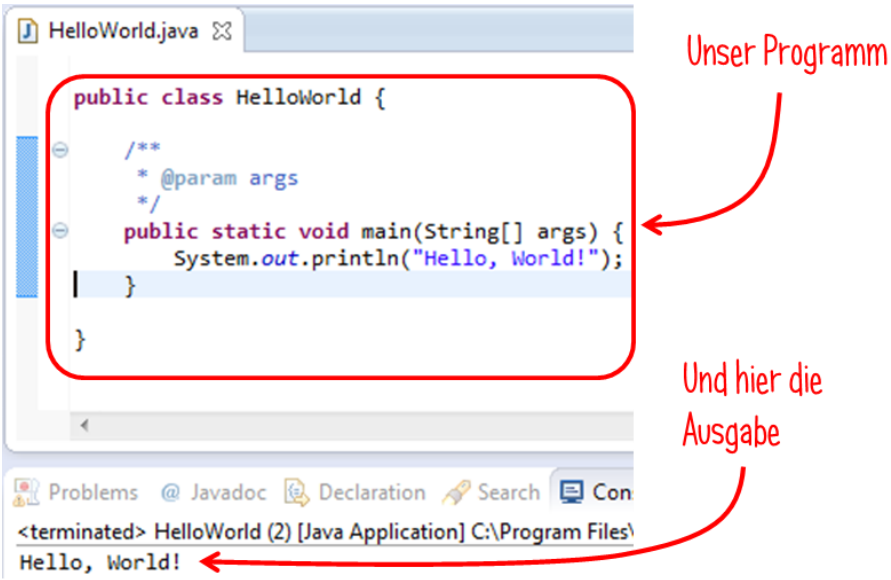
Dein Code sollte nun wie folgt aussehen:



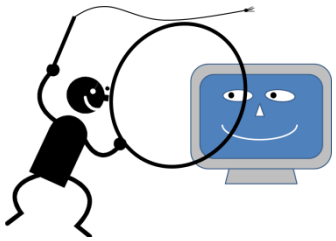
Was bedeutet diese Codezeile nun? Wir benötigen einen Befehl, der eine Konsolenausgabe mit dem Text „Hello, World!“ macht. Dieser Befehl heißt „println“. Wie (fast) alle Funktionalitäten in Java, handelt es sich hier um eine „Methode“ – also eine Funktion, welche in einer Klasse(nbibliothek) zu finden ist, und ein netter Mensch für uns irgendwann mal geschrieben hat. Wir müssen nun nur noch diese Methode aufrufen. Damit die Methode weiß, was sie auf die Konsole schreiben soll, gibt man ihr ein Argument mit. In unserem Fall ist dies die Zeichenkette (oder auch „String“ genannt) „Hello, World!“. Alles zwischen den beiden Anführungsstrichen wird somit als eine Zeichenkette interpretiert und der Methode als Argument übergeben.

Nachdem es schier unzählige Methoden gibt, müssen diese irgendwie strukturiert werden. Dies erfolgt in Bibliotheken und Klassen. Es gibt einige Standardbibliotheken – eine davon beinhaltet die Klasse „System“ in der verschiedene Elemente zu finden sind – unter anderem der „Standard Output Stream“. Dies ist ein Datenstrom, welcher für alle Textausgaben genutzt wird, welche nicht detaillierter spezifiziert sind. Dieser Datenstrom beinhaltet nun unter anderem die Methode „println“.

So – nun können wir unser kleines Programm testen. Wir starten in Eclipse mit dem Knopf unser Programm und siehe da – in dem unteren Konsolenfenster sehen wir „Hello, World!“.



3 Mündiger User

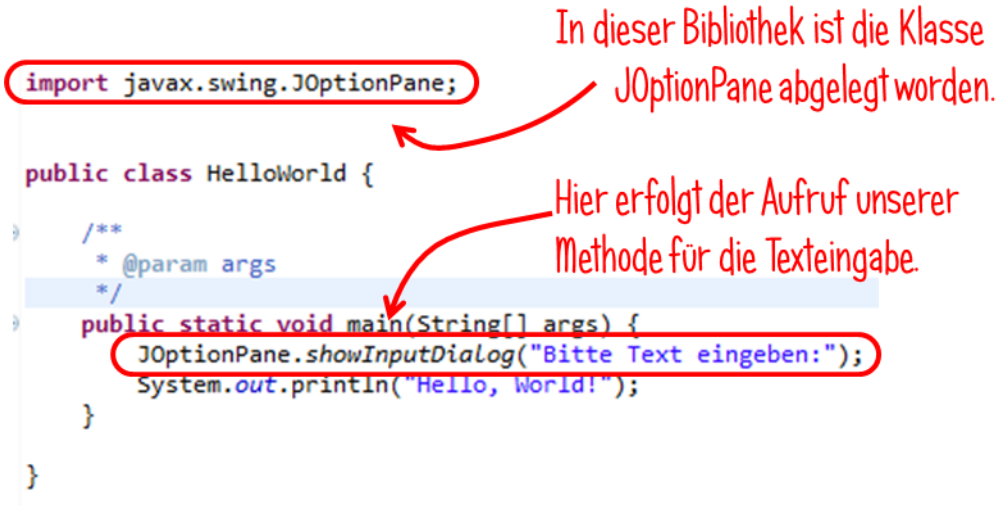


Der erste Schritt ist also getan – wir haben tatsächlich den Computer dazu gebracht nach unserer Pfeife zu tanzen. Wir haben gesagt „gebe etwas aus“ und der Rechner hat etwas ausgegeben. Soweit ein Erfolg! Als Problem könnte sich allerdings herausstellen, dass das Programm wirklich immer nur das gleiche machen kann. Und zwar „Hello, World!“ ausgeben. Eine Userinteraktion ist komplett unter den Tisch gefallen. Programme leben aber davon, dass der User etwas am Rechner macht und anschließend das Programm sich den Wünschen des Users anpasst. Insofern müssen wir an dieser Stelle noch etwas ergänzen – wir müssen versuchen eine Usereingabe zu berücksichtigen. Da wir noch am Anfang unserer Programmiererkarriere stehen, legen wir die Latte relativ niedrig an und verlangen von unserem Rechner lediglich, dass er das Ausgibt, was wir vorher eingegeben haben. Zugegeben – mit diesem Programm werden wir wohl kein ernsthafter Konkurrent von Microsoft werden, aber wir fangen ja erst an!

Zuerst müssen wir herausfinden welche Möglichkeiten uns Java zur Verfügung stellt, Eingaben zu tätigen. Eine Internetrecherche wird uns hier sehr viele Optionen aufzeigen. Wir werden uns auf die für uns momentan einfachste Möglichkeit beschränken, indem wir ein einfaches Eingabefenster erzeugen, in welches wir einen beliebigen Text eingeben. Dies wird über folgenden Befehl realisiert:

```
JOptionPane.showInputDialog("Bitte Text eingeben:");
```

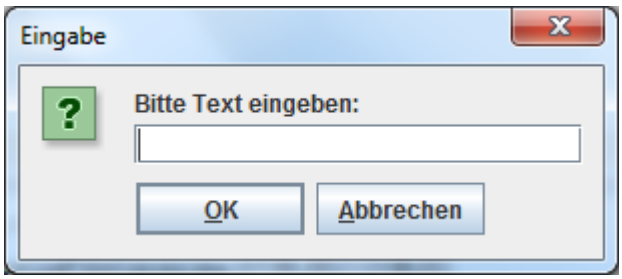
Ihr ahnt es bereits – auch hier nutzen wir wieder eine vordefinierte Klasse, namens „JOptionPane“. Diese wiederum hat unter anderem eine Methode, namens „showInputDialog“. Sie sorgt dafür, dass uns Java ein Texteingabefeld zeigt, in der wir etwas reinschreiben können. Die Methode erwartet einen Parameter, welchen wir mit der Zeichenkette „Bitte Text eingeben:“ füllen. Unser Code sollte nun wie folgt aussehen:



Eclipse war so frei und hat uns automatisch eine neue Zeile eingefügt, welche mit dem Schlüsselwort „import“ beginnt (sollte Eclipse das nicht getan haben, so ist eine Einstellung falsch – bitte in diesem Fall die Importzeile per Hand nachtragen). In diesem Statement wird dem späteren Interpreter (bzw. eigentlich Classloader) mitgeteilt, wo er die Klasse „JOptionPane“ findet. Es ist eine Referenz auf eine Java Klassenbibliothek – in unserem Fall findet sich die Klasse in der Bibliothek „javax.swing“.

Zusatzinfo:
Swing ist eine Bibliothek, welche sich primär um GUI (also Graphical User Interface) Programmierung kümmert – also Programme, die ein Dialogfenster aufweisen. Wir nutzen aus dieser Bibliothek nur ein kleines „Abfallprodukt“ und zwar ein kleines Fenster mit einer Eingabemaske. Es gibt neben Swing noch andere Bibliotheken für die GUI Programmierung (bspw. SWT – Standard Widget Toolkit), welche andere Schwerpunkte aufweisen. Für uns ist allerdings Swing die wichtigste Bibliothek für GUIs, da man immer davon ausgehen kann, dass die Bibliotheken auf einem Java Rechner vorhanden sind.

Sehen wir uns nun mal an, was passiert, wenn wir das Programm starten. Wie ihr seht, poppt tatsächlich eine Eingabemaske auf, in der „Bitte Text eingeben:“ steht. Diese beinhaltet auch noch das Eingabefeld und zwei Buttons. Somit haben wir schon mal die erste Frage geklärt, wofür der Parameter in showInputDialog genutzt wird. Es handelt sich hier einfach nur um den Text, der in der Eingabemaske angezeigt werden soll.



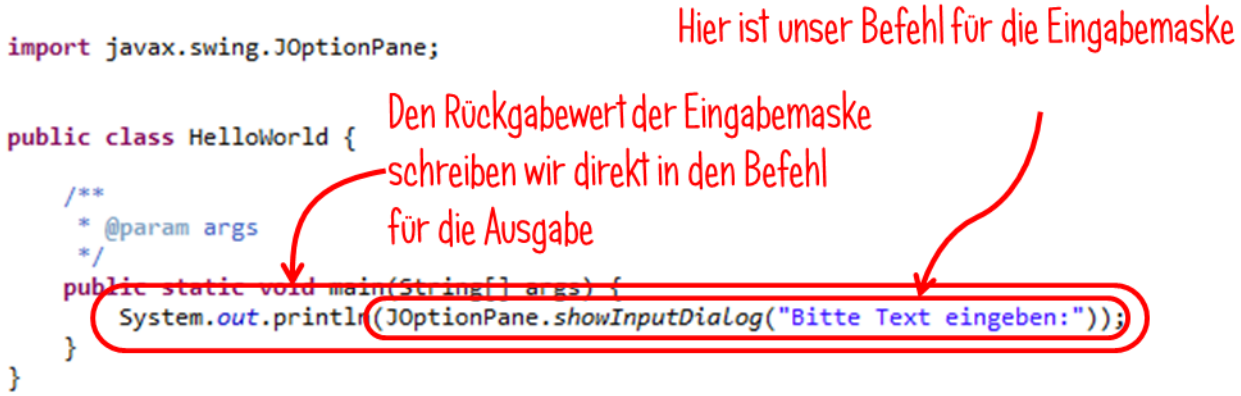
Wunderbar! Nun geben wir einen Text ein und klicken den OK Button. Doch was müssen wir feststellen? Wenn wir den Text „Hurra“ eingeben, zeigt die Konsole immer noch „Hello, World!“ an. Warum?

Ihr erinnert euch sicher an den eingangs erwähnten Grundsatz, dass wir dem Rechner alles haarklein vorgeben müssen, bevor er irgendetwas Sinnvolles macht. Wenn wir uns unseren Code nochmal genauer ansehen, dann müssen wir leider erkennen, dass wir zum einen dem Rechner nicht mitgeteilt haben, was er mit dem eingegebenen Text machen soll und zum anderen steht immer noch unsere Zeile:

```
System.out.println("Hello, World!");
```

im Code. Woher soll der arme Computer den wissen, was er eigentlich tun soll!?! Unser Ziel war es ja, dass wir den eingegebenen Text auf dem Bildschirm ausgeben wollen. Dazu müssen wir erst mal verstehen, wo unser eingegebener Text eigentlich landet. Wir geben ihn mühsam in die Eingabemaske ein, aber was passiert dann mit dem Text, wenn wir auf OK klicken? Das Fenster verschwindet – unser Text denn etwa auch? Nun, wenn sich die Java Autoren irgendetwas beim Erschaffen der Methode „showInputDialog“ gedacht haben sollten, dann müssen wir davon ausgehen, dass der Text noch irgendwo herumfliegt. Nur wo?

Hierzu müssen wir uns nochmal mit den Methoden beschäftigen. Eine Methode (in prozeduralen Programmiersprachen eine Funktion) ist ein Stück Programm, welches Daten entgegennehmen und Daten wieder an den Aufrufer zurückschicken kann. Das Daten entgegennehmen kennen wir bereits – das sind die Parameter, welche in Java zwischen zwei Runden Klammern nach dem Methodennamen stehen. Bei der Methode „println“ war dies der Text, welcher ausgegeben werden soll. Für unseren eingegebenen Text ist es jedoch nun von Interesse, wie die Methode den eingegebenen Wert nun wieder ausspuckt. Dies geschieht über den Rückgabewert. Man kann es sich so vorstellen, dass sobald die Methode ihre Arbeit verrichtet hat, die Methode „zum Rückgabewert“ wird. Am besten sieht man es, wenn man unseren Code wie folgt ändert:



Wenn wir nun den Code ausführen sehen wir, dass unser eingegebener Text tatsächlich auf der Konsole erscheint. Was ist also passiert? Der Computer arbeitet unser Konstrukt von innen nach außen ab – also er beginn unsere Codezeile mit dem Befehl „showInputDialog“. Daraufhin zeigt er den Dialog an, wir geben unseren Text ein und klicken auf „OK“. Der von uns eingegebene Text wird nun von der Methode „zurückgegeben“, was so viel heißt als dass der gesamte Text JOptionPane.showInputDialog(„Bitte Text eingeben:“) quasi durch den von uns eingegebenen Text ersetzt wird. Geben wir bspw. „Hallo“ ein, so würde nach dem Klick auf OK der Rechner folgenden Code interpretieren:

```
System.out.println("Hallo");
```

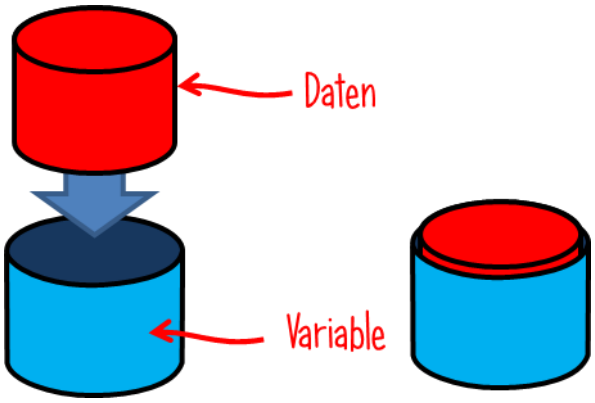
Er würde also das von uns eingegebene wiederum als Argument des println Befehls verwenden. Dadurch würde somit das „Hallo“ auf der Konsole ausgegeben werden. Man kann also die einzelnen Methoden ineinander verschachteln.

Wir halten also fest:

- Die Entwicklungsumgebung, welche wir nutzen ist Eclipse. Diese fasst (wie die meisten anderen Entwicklungsumgebungen auch) die einzelnen Files in Projekten zusammen.
- In Java werden die einzelnen Funktionalitäten in Klassen abgelegt, welche eine Art Bauplan für die im Speicher aktiven Programme darstellen.
- Innerhalb einer Klasse legen wir den Code, der beim Klassenaufruf beim Programmstart zuerst ausgeführt werden soll, in die „main“ – Methode ab. Diese Methode ist somit der Einstiegspunkt für die Java Virtual Machine (JVM) – also die Laufzeitumgebung von Java.
- Vorgefertigte Funktionen liegen in Bibliotheken und dort in eigenen Klassen. Die Bibliotheken müssen mitunter über den „import“ Befehl spezifiziert werden, damit die JVM die richtige Bibliothek lädt.
- Methoden sind gekapselte Funktionalitäten, welche einen Namen haben, eventuell Parameter besitzen und eventuell einen Wert zurückgeben. Wir haben eine Methode zur Textausgabe auf der Konsole und eine Methode zum Eingeben von Text durch den User kennengelernt.
- In Java wird jeder Befehl durch ein Semikolon „abgeschlossen“ (es sei denn er besitzt einen Rumpf, welcher bspw. mit geschweiften Klammern begrenze wird {})

4 Variablen – oder wie halte ich Daten fest

Soweit – so gut. Wir haben nun tatsächlich dem User ermöglicht, das Programm zur Laufzeit verschiedene Dinge machen zu lassen – auch wenn es nur unterschiedlicher Text ist, der ausgegeben wird. Die Frage ist nun, was kann man tun, wenn wir bspw. den Text zweimal hintereinander ausgeben wollen, und zwar auch dann, wenn der User den Text nur einmal eingibt? Hier müssen wir uns nun etwas Neues überlegen – so wie wir das bis jetzt gemacht haben funktioniert das leider nicht! Wir müssen es schaffen, den Text der vom User eingegeben wurde (zumindest temporär) zu speichern. Hier kommen die Variablen ins Spiel…

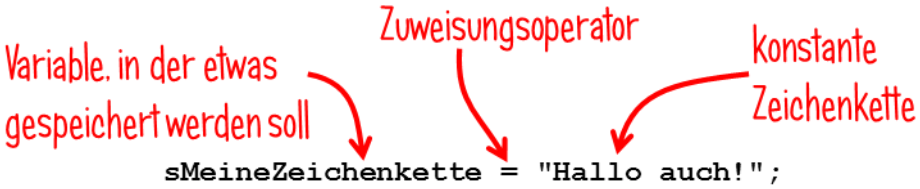


Variablen sind eine Art Behälter, in der wir Daten ablegen können. In Java kann jede Variable einen ganz bestimmten Typ von Daten aufnehmen. So gibt es Variablen für Zeichenketten, aber auch Variablen für ganze Zahlen. Solche Art der Variablennutzung nennt man „typisiert“. Es gibt allerdings auch Programmiersprachen, die nicht typisiert sind – in solchen Sprachen kann man in jeder Variable alles Mögliche ablegen – diese Sprachen (wie bspw. Javascript) sind nicht typisiert.

Aber wie kommt denn ein Datenwert in eine solche Variable – nun hierfür müssen wir erst mal eine Variable deklarieren. Deklarieren bedeutet, dass wir dem Computer sagen: „Hey, wir wollen gerne einen Datenwert abspeichern. Bitte sorge dafür, dass ein Platz im Speicher hierfür freigehalten wird. Die Art der Daten soll eine Zeichenkette sein. Und übrigens, die ich hätte gerne einen Namen für diese Variable, damit ich sie später im Programm wiederfinden kann.“

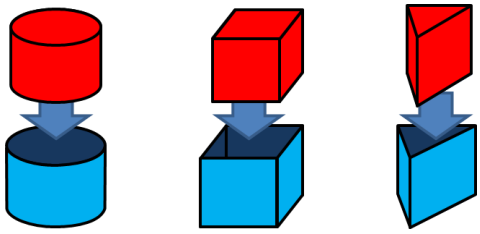
Technisch sieht dieser Befehl natürlich viel nüchterner aus. Wir benötigen ein Schlüsselwort für den Datentyp und einen (hoffentlich sinnvollen) Variablennamen.

Dieser Code veranlasst den Computer also, eine Variable für eine Zeichenkette im Speicher zu reservieren und versieht diese Variable mit einem Namen. Diese Variable existiert nur innerhalb der beiden geschweiften Klammern, in denen die Deklaration durchgeführt wurde (was derzeit die Klammern der „main“ Methode sind). Außerhalb dieser Klammern kann mit dieser Variablen nicht gearbeitet werden! Nun können wir einen Wert in diese Variable einspeichern. Dies geschieht über den sogenannten Zuweisungsoperator. Er veranlasst den Computer, in dem für die Variable reservierten Speicherplatz einen Wert abzulegen. Dieser Zuweisungsoperator ist einfach nur ein Istgleichzeichen. Wichtig – die Zuweisung erfolgt von rechts nach links. Also alles, was auf der rechten Seite des „=“ steht, wird in die Variable auf der linken Seite geschrieben:



Nach dieser Operation steht also in der Variable mit dem Namen „sMeineZeichenkette“ der Text „Hallo auch!“.

~~Blöde~~ Frage:
Was passiert eigentlich, wenn ich versuche Daten in einer Variablen abzulegen, welche vom Datentyp her nicht dem der Variablen entsprechen – also bspw. eine Zahl in eine Variable für Zeichenketten?
Antwort: Bei typisierten Programmiersprachen wird dies im Regelfall zu einem Fehler führen – meist schon beim Compillieren. Es gibt jedoch Sonderfälle – diese werden wir weiter Unten klären, beim Thema „Typecast“.



Jetzt haben wir so viel über Datentypen gehört – es wird wohl Zeit, sich diese einmal etwas genauer anzusehen. Grundsätzlich unterscheiden wir zwischen einfachen und zusammengesetzten Datentypen. Beginnen wir zuerst mal mit den einfachen Datentypen. Diese zeichnen sich dadurch aus, dass sie genau einen Wert ablegen können. Diese Werte können entweder:

- Ganze Zahlen
- Gleitkommazahlen
- Einzelne Zeichen
- Logische Werte (sog. „boolsche“ Werte sein – also wahr/falsch)

sein. Alle anderen Datenformate, wie bspw. Zeichenketten sind aus diesen einfachen Datentypen aufgebaut.

~~Blöde~~ Frage:
Ich dachte, ein Computer kann nur Zahlen speichern – schließlich liegen im RAM nur Einsen und Nullen, welche als Zahlen zu interpretieren sind. Wieso also jetzt auf einmal Zeichen und boolsche Werte?
Antwort: Es ist richtig, dass Computer nur Zahlen speichern. Ein Zeichen ist jedoch nichts anderes als eine Zahl, der ein Zeichen zugeordnet wird. Das kleine ‚a‘ ist bspw. die Zahl 97. Wenn ein Computer nun eine Variable für Zeichen sieht, in der die Zahl 97 gespeichert ist, dann würde er bei der Ausgabe einfach ein ‚a‘ ausspucken.

Werfen wir nun mal einen Blick auf das Innerste des Rechners, dem Speicher. Dort müssen ja früher oder später die Werte landen. Die meisten wissen ja, dass in einem Computerspeicher die kleinste adressierbare Einheit ein „Byte“ ist – also eine Ansammlung von acht Bit. Alle Werte, welche der Rechner nun ablegen möchte, muss er irgendwie in dieses Raster bringen. Wenn wir kurz überschlagen, dann finden wir heraus, dass der Rechner in einem Byte $2^8 = 256$ Werte ablegen kann. Eigentlich sind dies die Zahlen 0 bis 255, wobei der Rechner sich vorbehält diese Werte anders zu interpretieren.

Das erste Problem, das sich einstellt ist die Frage des Vorzeichens. Man könnte jetzt bspw. einfach sagen, dass das erste Bit = 0 positiv und das erste Bit = 1 negativ bedeutet und alle anderen Bits somit für die Zahl zuständig wären, womit wir von -127 bis + 127 alle Werte darstellen könnten (8 Bit -> 1 Bit Vorzeichen und 7 Bit die Zahlen, würde +/- 2^7 = +/- 128 Kombinationen ergeben, also +0 bis 127 und -0 bis -127). Man sieht hier aber schon, dass wir die 0 zweimal hätten. Das passt aber nicht wirklich in unseren Erfahrungsschatz – es gibt nur eine 0. Also wird dieser Ansatz etwas anders dargestellt – man nutzt die Darstellung im „Zweierkomplement". Wir stellen uns folgende Situation vor – wir haben für unser Beispiel „nur" vier Bit (damit wir nicht so viele Kombinationen berücksichtigen brauchen) und müssen damit unser Zahlensystem aufbauen (Details zum Zahlensystem bspw. unter http://www.ulthryvasse.de/index.html) .

Viermal die 0 ist die Darstellung für die Zahl 0:

3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
0	0	0	0		0

Nun zählen wir 1 dazu, was uns zur Zahl 1 führt:

3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
0	0	0	1		1

Das können wir nun weiterführen, bis wir die ersten drei Bits mit 1 belegt haben

3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
0	0	0	0		0
0	0	0	1		1
0	0	1	0		2
0	0	1	1		3
0	1	0	0		4
0	1	0	1		5
0	1	1	0		6
0	1	1	1		7

Nun müssen wir uns konzentrieren! Wir gehen jetzt davon aus, dass wir die Hälfte unseres Zahlenbereiches genutzt haben. Die andere Hälfte müsste nun für die negativen Zahlen reserviert sein – richtig? Nun die alles entscheidende Frage – was ist -1 + 1? Wer jetzt nicht auf 0 kommt sollte dringend eine Pause machen, sich einen Kaffee einschenken und in einer halben Stunde weitermachen! Also, minus 1 plus 1 ergibt 0. Wenn wir uns jetzt wieder unsere Tabelle ansehen müssen wir uns fragen, wie müssen die Bits stehen, damit viermal die 0 herauskommt, wenn wir 1 hinzuzählen? Richtig, viermal die 1:

3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
1	1	1	1		-1

Nun zählen wir die 1 hinzu (und kehren den Übertrag unter den Tisch, da wir ja nur vier Bits haben):

3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
0	0	0	0		0

Basierend auf diesen Überlegungen können wir nun die Negativen Zahlen fortsetzen:

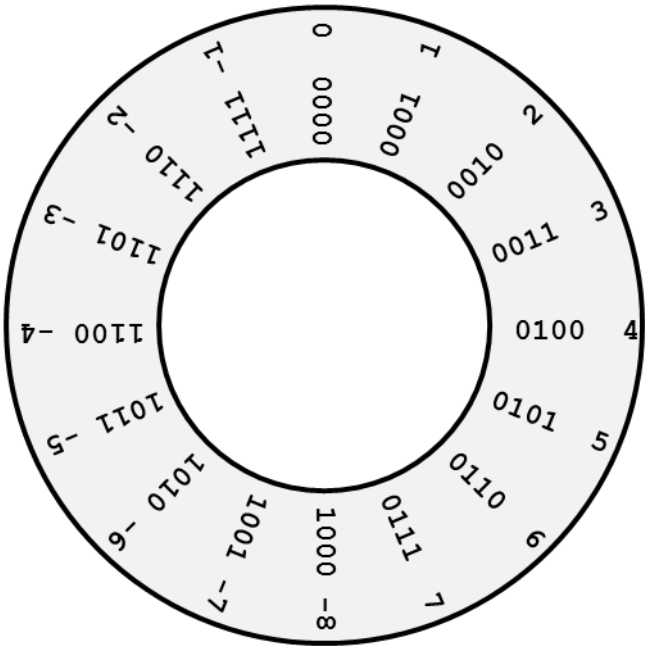
3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
1	1	1	1		-1
1	1	1	0		-2
1	1	0	1		-3
1	1	0	0		-4
1	0	1	1		-5
1	0	1	0		-6
1	0	0	1		-7
1	0	0	0		-8

Die Richtigkeit können wir prüfen, indem wir ausgehend von einer negativen Zahl immer eins hinzuzählen und daraufhin auf die nächsthöhere Zahl kommen. Hieraus ergeben sich folgende Konsequenzen:

- Es gibt eine negative Zahl mehr als positive, da die 0 als positiv gewertet wird.
- Wenn wir bei vier Bit zur höchsten positiven Zahl eins hinzuzählen, kommen wir bei der kleinsten (negativen) Zahl an.

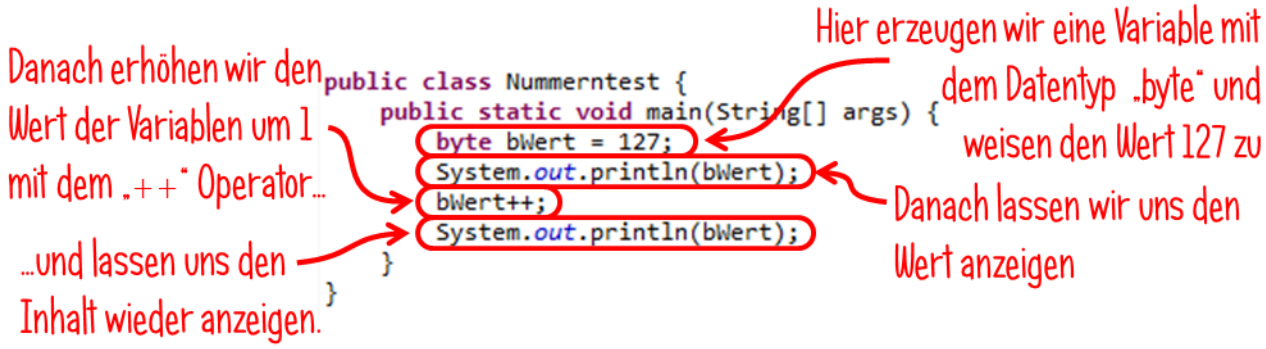
Grafisch können wir es uns wie folgt vorstellen:

3.Bit	2.Bit	1.Bit	0.Bit		Zahl:
0	0	0	0		0
0	0	0	1		1
0	0	1	0		2
0	0	1	1		3
0	1	0	0		4
0	1	0	1		5
0	1	1	0		6
0	1	1	1		7
1	0	0	0		-8
1	0	0	1		-7
1	0	1	0		-6
1	0	1	1		-5
1	1	0	0		-4
1	1	0	1		-3
1	1	1	0		-2
1	1	1	1		-1



Das Geniale daran ist nun, dass wir sehr einfach addieren und subtrahieren können, ohne immer auf das Vorzeichen achten zu müssen. Probier es aus!

Um nun zu beweisen, dass in Java es tatsächlich so aussieht, schreiben wir uns ein kleines Programm. Anders als in unserem „Papier und Bleistift“ Beispiel können wir hierfür keine Variablen mit 4 Bit verwenden (die kleinste Einheit war ja 8 Bit), sondern einen Datentyp namens „byte“, der – ihr ahnt es bereits – genau 8 Bit belegt und somit 2^8 Möglichkeiten bietet. Entsprechend unserer Überlegungen können wir somit Theoretisch den Zahlenraum von -128 bis +127 darstellen. Prüfen wir das mal kurz mit einem kleinen Programm:



Gehen wir den Code kurz durch. Zuerst deklarieren wir wieder eine Variable, namens „bWert“ vom Typ „byte“. In der gleichen Zeile schreiben wir auch direkt den initialen Wert hinein – diesen haben wir mit unserem größten möglichen Wert festgelegt, nämlich 127. Zur Kontrolle geben wir den Wert in der nächsten Zeile aus. Danach erhöhen wir den Wert der Variablen um 1. Dies geschieht mit dem „Inkrementoperator ++“ (Operatoren werden wir weiter Unten näher untersuchen). Wenn unsere Theorie nun richtig ist, müsste der Wert von Byte nun auf -128 springen. Probiert es aus – tippt den Code ein und lasst ihn laufen. Wie ihr seht, gibt die erste println Zeile die 127 aus und die zweite wie erwartet die -128.

Wie sieht das also nun im Speicher aus – wenn wir uns die Variable bWert mal genauer ansehen? Nehmen wir uns den Code nochmal zur Brust und blicken parallel in den Speicher unseres Computers. Auf der rechten Seite der folgenden Grafik sehen wir den Code, links jeweils einen Ausschnitt des Speichers, der momentan mit irgendwelchen zufälligen Werten aufgefüllt ist, wobei sie nur aus unserer Sicht „zufällig“ sind. Der

Computer könnte uns mit Sicherheit exakt erklären, warum die einzelnen Werte so sind, wie sie sind. Alle grau hinterlegten Speicherzellen sind von anderen Variablen oder Programmteilen belegt, die weiß hinterlegten stehen derzeit zur freien Verfügung. Gehen wir den Code einfach mal durch:

```
public class Nummerntest {
    public static void main(String[] args) {
        byte bWert = 127;
        System.out.println(bWert);
        bWert++;
        System.out.println(bWert);
    }
}
```

Am Anfang wurde die Variable noch nicht deklariert. Insofern belegt sie keinen Speicherplatz.

11111100	10010101	10010110
10110010	00011010	00000001
10010101	10110010	11001001
11000110	10011010	00110101

Nun lassen wir den Interpreter eine Zeile weiter gehen und ausführen:

```
public class Nummerntest {
    public static void main(String[] args) {
        byte bWert = 127;
        System.out.println(bWert);
        bWert++;
        System.out.println(bWert);
    }
}
```

Nun wird die Variable deklariert – also ein Speicherplatz reserviert und der Wert 127 hineincodiert.

11111100	10010101	10010110
10110010	00011010	00000001
10010101	01111111	11001001
11000110	10011010	00110101

Wir sehen, dass wie im oberen Bereich beschrieben, die interne Darstellung von 127 ganz normal nach dem Binärprinzip erfolgt. Nun kommt der nächste Schritt (Ausgabe lassen wir bei der Analyse mal aus) – wir erhöhen also den Wert um 1.

```
public class Nummerntest {
    public static void main(String[] args) {
        byte bWert = 127;
        System.out.println(bWert);
        bWert++;
        System.out.println(bWert);
    }
}
```

Die Erhöhung läuft nun streng mathematisch – also einfach binär weiterzählen.

11111100	10010101	10010110
10110010	00011010	00000001
10010101	10000000	11001001
11000110	10011010	00110101

Wir sehen, dass Im Speicher nun die 10000000 steht – was binär eigentlich 128 wäre, in der Interpretation durch Java aber als -128 ausgegeben wird, da das linke Bit mit 1 gesetzt wurde und somit eine negative Zahl vorliegt.

Puh – das wirkt auf den ersten Blick ganz schön kompliziert. Nun kommt aber die gute Nachricht – wir müssen von diesen Interna eigentlich überhaupt nichts wissen. Java kümmert sich selbstständig um diese Details. Wir müssen lediglich zu den entsprechenden Datentypen wissen, was der kleinst- und größtmögliche Wert ist. Das ist alles! Jetzt fragt ihr euch sicher, warum ich mir die Mühe gemacht habe, das alles in dieses Dokument zu schreiben. Die Antwort ist – damit ihr seht, dass im Rechner alles mit rechten Dingen vorgeht und ihr verstehen könnt, warum bei einer „byte“ Variablen die Rechnung 127 + 1 die Zahl -128 ergibt.

Dann wollen wir die einzelnen Variablen mal Stück für Stück durchgehen. Beginnen wir mit den Variablen für ganzzahlige Werte. Hier bietet uns Java einen ganzen Strauß an Datentypen:

Ganzzahlige Werte:

Datentyp:	Speicherplatz:	Kleinster Wert:	Größter Wert:
byte	1 Byte	-128	127
short	2 Byte	-32.768	32.767
int	4 Byte	-2.147.483.648	2147.483.647
long	8 Byte	-922.3372.036.854.775.808	922.3372.036.854.775.807

Üblicherweise verwenden wir den Datentyp int, wenn wir ganzzahlige Werte benötigen. Java ist auf diesen Datentypen optimiert. Nur wenn wir ganz viele Variablen mit ganzen Zahlen (oder besser ganz viele

ganzzahlige Werte) benötigen, wie etwa bei einem Datenstrom, nutzen wir kleinere Einheiten wie bspw. „byte“. Long wird nur dann genutzt, wenn wir größere Zahlen mit 100%iger Genauigkeit brauchen, als int dies ermöglicht. Dies kommt aber relativ selten vor.

Die nächste Art von Daten, welche wir uns ansehen wollen sind die Gleitkommazahlen. Bei einer Gleitkommazahl werden anders als bei ganzen Zahlen zwei Informationen geliefert, die sogenannte Mantisse und der Exponent (im Regelfall zur Basis 10). Die Zahl 1234,5678 wird in der Exponentenschreibweise also wie folgt dargestellt: $1{,}2345678 * 10^3$. Die Mantisse ist also die Zahl 1,2345678 und bestimmt somit die Genauigkeit oder „Präzision“ der Zahl (man spricht hier auch von Anzahl signifikanter Stellen). Der Exponent die 3 und bestimmt die eigentliche Größe oder Dimension der Zahl. Wenn in Java nun die beiden Zahlen getrennt abgelegt werden, dann kann man die Genauigkeit und die Größe getrennt voneinander angeben. Folgende Möglichkeiten existieren:

Gleitkommawerte:

Datentyp:	Speicherplatz:	Kleinster Absolutwert:	Größter Absolutwert:	Anzahl Stellen:
float	4 Byte	$+/- 1.4 * 10^{-45}$	$+/- 3.4 * 10^{38}$	7
double	8 Byte	$+/- 4{,}9 * 10^{-324}$	$+/- 1{,}7 * 10^{308}$	15

Wer mehr über die Genauigkeit von single und double precision Zahlen wissen möchte, der sei auf das Internet verwiesen. Einfach nach der IEEE 754-1985 Norm suchen…

Im Regelfall verwenden wir beim Programmieren in Java die double Werte, da die JVM mit diesem Datenformat sehr gut umgehen kann. Auch hier gilt – wenn wir eine hohe Anzahl an Daten verarbeiten, dann sollte wir uns überlegen, auf float umzustellen. Dies wird oftmals bei Grafikapplikationen gemacht – man nutzt hier für die einzelnen arithmetischen Prozeduren float Werte.

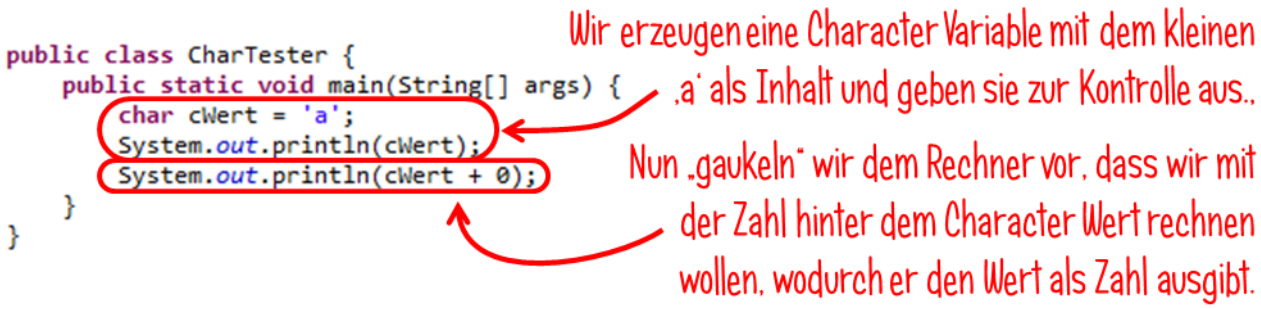
Das nächste sehr spannende Thema sind die einzelnen Zeichen. Man spricht hier von „Character“ Werten. Wie weiter Oben schon erwähnt, sind Buchstaben für den Rechner nichts anderes als Zahlenwerte, welche als Buchstabe interpretiert werden. Man kann den Computer auch dazu bringen, die Zahl hinter dem Buchstaben preiszugeben. Aber fangen wir erst mal mit den Eckdaten unseres neuen Datentyps an:

Einzelne Zeichen:

Datentyp:	Speicherplatz:	Kleinster Wert:	Größter Absolutwert:
char	2 Byte	0 (nicht druckbar)	65535 (nicht druckbar)

Die gute Nachricht ist, dass Character Werte keine negativen Zahlen beinhalten. Das heißt, wir finden im Speicher die Zahlen 00000000.00000000 bis 11111111.11111111, welche im Dezimalsystem entsprechend 0 bis 65535 ergeben. Die Character Werte sind also in 16 Bit codiert – sie sind also auf UTF-16 ausgelegt, sprich Unicode Transformation Format mit 16 Bit.

Um nun den Zahlenwert einer Character Variable anzuzeigen, nutzen wir folgenden Code:



Nun lassen wir unser Programm laufen und sehen uns die Konsolenausgabe an. Wir sehen zuerst das ‚a‘ und eine Zeile darunter die Zahl 97. Dies ist genau der Wert, der in den Zeichentabellen für das kleine ‚a‘ steht. Dies könnt ihr bspw. in folgendem Link nachvollziehen: http://de.wikipedia.org/wiki/Ascii#ASCII-Tabelle

Wenn ihr schon mal auf dieser Seite seid, dann könnt ihr euch gleich mal die Anordnung der Zeichen ansehen. Auffällig ist, dass die Zahlen 0-9, die Zeichen A-Z und a-z in einer Reihe liegen. Diesen Umstand können wir uns später nochmal zu Nutze machen, bspw. für Zeichenvalidierungen.

Der letzte einfache Datentyp ist „boolean“. Dieser ist recht einfach gestrickt – er kennt nur zwei Zustände, true und false. Wofür braucht man nun einen solchen Datentyp? Boolean Werte sind wichtig für die Darstellung von Zuständen. Wenn eine Methode bspw. zwei Werte vergleicht, dann gibt sie entweder „true“ oder „false“ zurück – was ein „boolean“ Wert ist. Wir werden bei Schleifen und Verzweigungen darauf nochmal näher eingehen. An dieser Stelle nun erst mal die Eckdaten von boolean:

Wahrheitswert:

Datentyp:	Speicherplatz:	Wert für „falsch“:	Wert für „wahr“:
boolean	1 Byte	false	true

~~Blöde~~ Frage:
Eine Variable, welche nur zwei Zustände haben kann müsste doch eigentlich nur ein Bit Speicherplatz benötigen und nicht 8?
Antwort: Das ist prinzipiell richtig – zwei Zustände können mit einem Bit zwar dargestellt werden, aber ein einzelnes Bit kann nicht adressiert werden. Insofern müssen wir hier in den sauren Apfel beißen und 7 Bit einfach ungenutzt liegen lassen. Aber da boolean Werte nicht exzessiv genutzt werden ist das durchaus verschmerzbar.

Wir halten also fest:

- Einfache Datentypen belegen einen festgelegten Speicherbereich.
- Der Speicher ist in einzelne Elemente a 8 Bit (also 1 Byte) aufgeteilt, insofern belegt jeder Datentyp ein vielfaches von 8 Bit
- Es gibt Datentypen für ganze Zahlen, Gleitkommazahlen, einzelne Zeichen und Wahrheitswerte („boolsche“ Werte)
- Jede Variable, die in Java genutzt wird, muss vorher deklariert werden, indem der Datentyp und der Variablenname angegeben wird. Dadurch wird der Speicherplatz für die Variable reserviert und mit einem Namen versehen, damit man mit der Variablen arbeiten kann.
- Die Variablen können nur innerhalb der geschweiften Klammern stehen, in denen sie deklariert wurden. Nach der schließenden Klammer gibt Java den Speicherplatz wieder frei.
- Eine Variable braucht auch einen initialen Wert – die Variable muss also mit einem Wert belegt werden. Vorher kann man mit der Variablen nicht arbeiten. Einen Wert bekommt man mit dem = Operator in die Variable, wobei die Variable immer links steht und der Wert, welcher abgelegt werden soll, steht immer rechts (dies kann auch eine Rechnung oder eine Variable sein).
- Jeder einfache Datentyp hat einen definierten Wertebereich. Wenn dieser überschritten wird, hat dies meist fehlerhafte bzw. für den User nicht nachvollziehbare Ausführungen zur Folge.

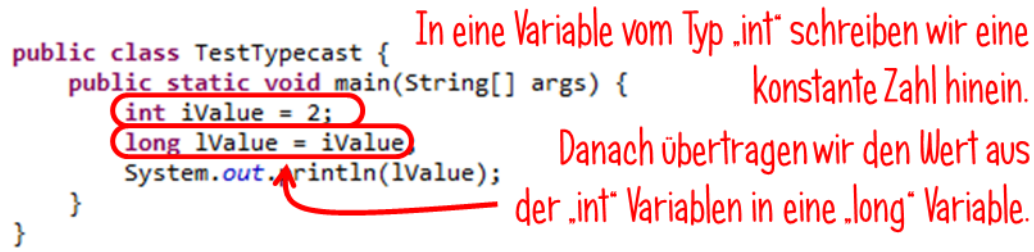
Wie wir gesehen haben, können wir Zahlenwerte in verschiedenen Datentypen ablegen. Bspw. können wir den Wert 2 in folgenden Datenformaten ablegen:

- byte
- short
- int
- long
- float
- double
- char (ja, auch das geht)

Hieraus ergeben sich zwei wesentliche Fragen. Wenn wir bspw. folgende Codezeile ansehen:

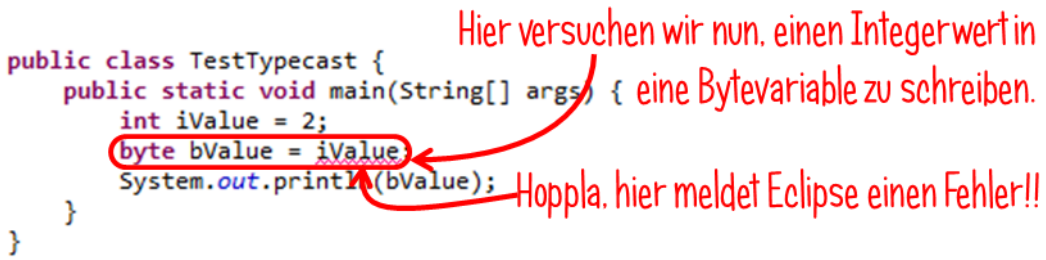
```
int iValue = 2;
```

stellt sich die Frage, woher weiß Java nun eigentlich, dass 2 vom Datentyp „int“ ist? Die Antwort ist, dass Java bei konstanten ganzzahligen Werten erst mal von int ausgeht. Bei Gleitkommazahlen übrigens von double. Wie kann man dann eigentlich einen initialen Wert in eine byte Variable schreiben, wenn die ganzzahlige Konstante vom Typ int ist? Dies ist leider nicht so einfach zu beantworten. Grundsätzlich spielt der sogenannte „Typecast“ eine große Rolle. Unter Typecast versteht man die Umwandlung des Datentyps, idealerweise ohne den Wert zu verändern. Sehen wir uns folgenden Code hierzu mal an:



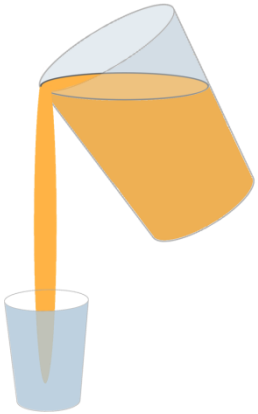

```
public class TestTypecast {
    public static void main(String[] args) {
        int iValue = 2;
        long lValue = iValue;
        System.out.println(lValue);
    }
}
```

Wie wir in der Ausgabe sehen, gibt das Programm tatsächlich die 2 aus. Der Wert wurde also vom Datentyp „int“ in einen Datentyp „long“ umgewandelt. Dies nennt man „**impliziten Typecast**“. Implizit, weil Java von alleine darauf kommt, dass der Integerwert in einer „long“ Variablen abgelegt werden kann. Bei impliziten Typecasts können wir immer davon ausgehen, dass keine Daten verloren gehen. Nun versuchen wir mal einen anderen Weg. Wir ändern unseren Code wie folgt ab:



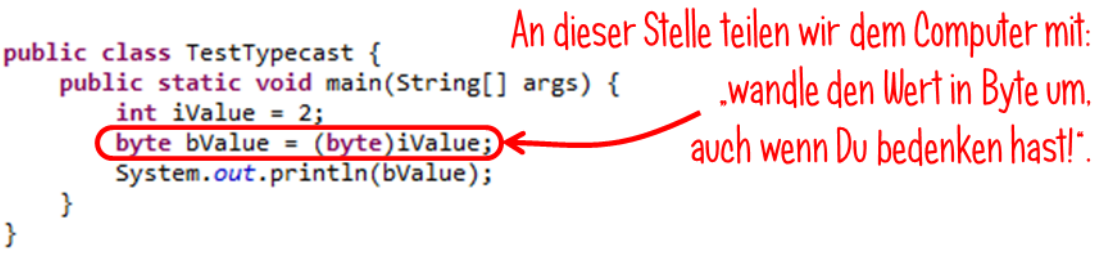

```
public class TestTypecast {
    public static void main(String[] args) {
        int iValue = 2;
        byte bValue = iValue;
        System.out.println(bValue);
    }
}
```

Was ist denn nun passiert – Java möchte offensichtlich nicht, dass wir einen Wert aus einer Integervariablen in eine Bytevariable schreiben. Warum eigentlich nicht – schließlich ist in der Integervariablen „nur“ eine 2, die in eine Bytevariablen ganz locker reinpasst…



Die Erklärung ist wie folgt. Der Compiler sieht an dieser Stelle nur, dass wir versuchen aus einer Variablen, welche einen Wertebereich von -2.147.483.648 bis 2147.483.647 haben kann den Wert (was auch immer dieser Wert ist) in eine Bytevariable zu schreiben, welche einen Wertebereich von -128 bis 127 hat. Da der Integerwert keine Konstante ist, sondern eine Variable, analysiert der Compiler den Inhalt nicht. Er geht vielmehr davon aus, dass hier ein potentielles Problem vorliegt und verweigert die Weiterarbeit. Es könnte ja sein, dass der Inhalt der Integervariablen größer ist, als das Fassungsvermögen der Bytevariablen!

Was ist aber nun, wenn ich genau weiß, dass in der Integervariablen ein Wert enthalten ist, der auf jeden Fall in die Bytevariable hineinpasst? Nun, in diesem Fall kann man dem Rechner explizit mitteilen, das der Datenübertrag von der großen in die kleinere Variable OK ist. Man nennt dies einen „**expliziten Typecast**“. Dieser wird durchgeführt, wenn der Zieldatentyp in Klammern vor den Quellwert geschrieben wird.




```
public class TestTypecast {
    public static void main(String[] args) {
        int iValue = 2;
        byte bValue = (byte)iValue;
        System.out.println(bValue);
    }
}
```

Nun ist die Eclipse Fehlermeldung auch verschwunden! Java akzeptiert nun unsere Operation und geht davon aus „Der Programmierer wird schon wissen, was er macht!“.

Wir halten also fest:

- Konstante Zahlenwerte haben einen festen Datentyp – Integer bei ganzen Zahlen und Double bei Gleitkommawerten.
- Wenn wir Daten von einer Variablen eines Datentyps in eine Variable eines anderen Datentyps kopieren, dann ist ein Typecast notwendig. Dieser ist entweder implizit oder explizit.
- Ein impliziter Typecast wird immer dann durchgeführt, wenn der Wertebereich des Zieldatentyps den Wertebereich des Quelldatentyps auf jeden Fall aufnehmen kann. Dies geschieht bei Variablen immer unabhängig vom tatsächlichen Dateninhalt.
- Bei impliziten Typecasts gehen keine Daten verloren.
- Ein expliziter Typecast muss immer dann durchgeführt werden, wenn der Wertebereich des Zieldatentyps den Wertebereich des Quelldatentyps nicht aufnehmen kann. Auch hier ist bei Variablen nicht der tatsächliche Inhalt, sondern der mögliche Inhalt entscheidend.
- Bei expliziten Typecasts können Daten verloren gehen, wenn der übertragene Wert im Zieldatentyp keinen Platz findet.
- Ein expliziter Typecast wird durchgeführt, indem der Zieldatentyp in Klammern vor den Quellwert geschrieben wird.

Sehen wir uns die Typecastmöglichkeiten nochmal strukturierter an. „=“ bedeutet, dass kein Typecast notwendig ist. „I“ steht für „impliziten Typecast“ und „E“ für „expliziten Typecast“.

Zieldaten-typ:	Wertebereich:							
byte	-128 bis 127	=	E	E	E	E	E	E
short	-32.768 bis 32.767	I	=	E	E	E	E	E
int	-2.147.483.648 bis 2.147.483.647	I	I	=	E	E	E	I
long	-9.223.372.036.854.775.808L bis 9.223.372.036.854.775.807L	I	I	I	=	E	E	I
float	+/-1,4E-45 bis +/-3,4E+38	I	I	I	I	=	E	I
double	+/-4,9E-324 bis +/-1,7E+308	I	I	I	I	I	=	I
char	0 bis 65.535	I	I	I	I	I	I	=
	Quelldatentyp:	btye	short	int	long	float	double	char

Gehen wir die Punkte kurz durch. Dass die kleineren Datetnypen wie bspw. „byte“ jeweils einen expliziten Typecast benötigen sollte klar sein – schließlich können sie nur den kleinsten Wertebereich fassen. Interessant ist, dass „char“ in dieser Tabelle ganz normal gehandhabt wird und nur dann einen expliziten Typecast benötigt, wenn der größte positive Wert des Zieldatentyps kleiner als 65.535 ist.

Weiterhin ist bemerkenswert, dass die beiden Gleitkommazahlen alle Datentypen mit ganzen Zahlen ohne Typecast aufnehmen können – dies bedeutet also, dass trotz der wenigen signifikanten Stellen es Java trotzdem gelingt, eine „long“ Variable in einem „float“ Datentyp verlustfrei abzulegen.

~~Blöde~~ Frage:
Was passiert denn nun eigentlich, wenn ich bspw. den Wert 200 aus einer Integervariable mit Hilfe eines expliziten Typecasts in eine Bytevariable schreibe, obwohl sie dort keinen Platz findet?
Antwort: Es werden die überschüssigen Bits einfach abgeschnitten und es kommt ein auf den ersten Blick zufälliger Wert heraus. Beim zweiten Blick erkennen wir, dass wir beim Überschreiten der 127er Marke die Werte wieder bei -128 beginnen und wir dann bei -56 landen. Es ist aber auf gar keinen Fall sinnvoll, einen expliziten Typecast zu machen wenn wir nicht 100% wissen, dass der Wert in den Zieldatentyp hineinpasst, oder wir genau diesen Datenverlust wollen! Dies ist bspw. dann der Fall, wenn wir von double nach int casten – was zur Folge hat, dass die Nachkommastellen einfach abgeschnitten werden.

Das letzte, was uns jetzt bei den einfachen Datentypen noch fehlt, ist die Frage, wie kann ich eigentlich in eine Variable vom Datentyp „byte“ oder „short“ einen konstanten Wert hineinschreiben, wenn die konstanten Zahlen in Java doch vom Typ „int“ sind. Hier kann ich euch mal wieder beruhigen – die Macher von Java haben es so hingebogen, dass der Compiler bei **konstanten Zahlenwerten** erkennt, ob der Wert noch hineinpasst oder nicht. Spannender ist die Frage, wie schaffe ich es in eine Variable vom Typ „long“ einen Wert hineinzuschreiben, der größer als 2.147.483.647 ist? Hier muss ich dem Compiler mitteilen, dass die Konstante vom Typ „long“ ist, indem ich ein „l“ oder „L“ an die Variable dranhänge:

```
long lValue = 5000000000000L;
```

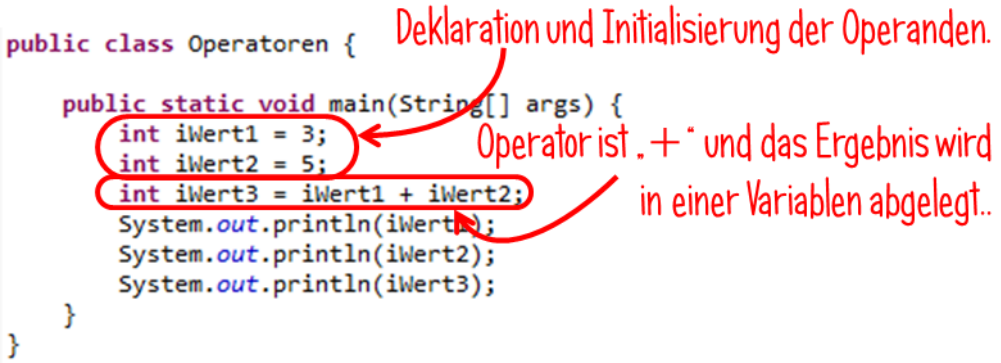
Eine ähnliche Lösung gibt es für die Werte von „float“ Variablen. Da konstante Gleitkommazahlen immer double sind, muss ich bei einer Zuweisung in „float“ Variablen ein „f“ hinzufügen:

```
float fValue = 3.4f;
```

5 Operatoren – mach endlich was mit den Daten!

Super – jetzt können wir eine Vielzahl von Werten im Rechner speichern und von einer Variablen in die andere kopieren. Was uns jetzt aber noch fehlt ist die Möglichkeit, die Daten tatsächlich zu verändern. Schließlich soll der Computer ja für uns arbeiten und Dinge mit den Daten tun. Wir benötigen also eine Möglichkeit, Operationen an den Werten auszuführen. Dies wird mit sogenannten Operatoren durchgeführt. Wir werden eine Vielzahl von Operatoren kennenlernen, von denen wir aber in der Praxis nur eine überschaubare Menge benötigen werden. In besonderen Situationen wirst Du aber froh sein wenn du dich daran erinnerst, dass du für diese besondere Situation irgendetwas mal gelesen hast und dich hoffentlich daran erinnerst und an der richtigen Stelle nachliest.

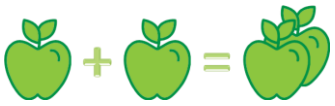
Fangen wir mal mit den offensichtlichen Operatoren an – den „**arithmetischen Operatoren**“. Hier unterscheiden wir zwischen solchen, die Inhalte der Operanden verändern und solchen, die die Inhalte nicht verändern. Folgendes Beispiel soll dies verdeutlichen:



In die Variable iWert3 wird das Ergebnis der Operation iWert1 + iWert2 abgelegt (Achtung – der „+“ Operator bei Strings hat eine andere Wirkung – das sehen wir uns später an). Wie wir bei der Ausgabe sehen, werden die Werte iWert1 und iWert2 nicht verändert – es steht immer noch 3 und 5 drin. Die Operatoren bleiben somit unangetastet. Lediglich in iWert3 steht ein neuer (bzw. initialer) Wert von 8 drin. Man muss jetzt kein Genie

sein um zu ahnen, dass die Operatoren „-“ für Subtraktion, „*“ für Multiplikation und „/“ für Division stehen und die Klammerregeln wie in Mathe gelten. Allerdings gilt es vor allem bei dem Thema Division noch eine Besonderheit zu beachten und zwar der Datentyp des Ergebnisses.
Der Computer verarbeitet die Werte in Registern, was nichts anderes ist als eine Art Speicher. Dort muss also ebenso eine Interpretation des Datentyps durchgeführt werden. Ergo muss auch das Ergebnis einer Rechnung einen Datentyp besitzen – unabhängig davon ob wir das Ergebnis in eine Variable schreiben oder gleich am Bildschirm ausgeben. Jeder Wert mit dem im Rechner gearbeitet wird hat zwangsläufig einen Datentyp. Die Frage ist nun, welchen Datentyp hat nun folgende Rechnung, bzw. was wird der Rechner ausgeben:

```
int iWert1 = 3;
int iWert2 = 2;
System.out.println(iWert1 / iWert2);
```



Wenn ihr das ausprobiert seht ihr, dass der Rechner den Wert „1“ ausgibt. Wie kommt das? Die Erklärung ist ganz einfach – Äpfel mit Äpfel verrechnet ergibt – na, wer kann es sich denken? Richtig: Äpfel! Wenn wir also mit einem arithmetischen Operator zwei Integerwerte verrechnen, kommt Integer heraus. Das eigentliche Ergebnis 1.5 kann nicht gespeichert werden, also wird die Nachkommastelle einfach abgeschnitten und das Ergebnis ist 1. Wenn ich das richtige Ergebnis haben möchte, so muss ich mindestens einen Wert in einen anderen Datentypen umwandeln – sagen wir „double“. Dies geschieht wie wir gelernt haben mittels Typecast:

```
int iWert1 = 3;
int iWert2 = 2;
System.out.println((double)iWert1 / iWert2);
```

Das wird oft übersehen! Wir müssen beim Hantieren mit Datentypen uns diese Regel immer wieder vorsagen:

Bei einer arithmetischen Operation ist der Datentyp des Ergebnisses immer gleich dem des „größten“ Operandendatentyps.

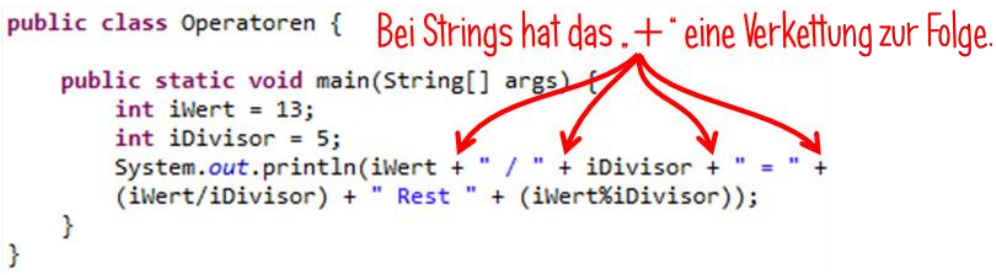
Seht euch mal folgenden Code an und versucht zu überlegen, was das Programm an dieser Stelle macht:

```
int iWert1 = 2;
int iWert2 = 3;
int iWert3 = 4;
System.out.println(iWert3 / (iWert1 / iWert2));
```

Na, erraten? Wenn nicht, dann den Code mal abtippen und staunen! Java meldet einen Fehler. Warum? Weil Java zuerst die Klammer rechnet (also iWert1 / iWert2) und das eigentliche Ergebnis 0.66 im Integerformat nur die 0 ist (die Nachkommastellen werden ja abgeschnitten). Division durch 0 ist aber nicht definiert und schon knallt es.

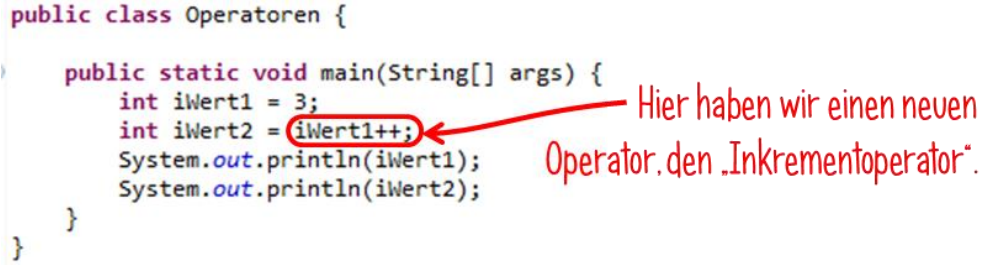
Einen grundlegenden arithmetischen Operator habe ich jedoch noch unterschlagen, die Modulorechnung. Dies ist definiert als „Rest einer ganzzahligen Division“. Folgendes Beispiel soll dies verdeutlichen:
13 / 5 soll errechnet werden. Hierzu prüfen wir, wie oft die 5 in 13 hineinpasst. Antwort: 2 mal. Der Rest, der hierbei übrig bleibt ist 3, da 2*5 = 10 und 13 – 10 = 3 ergibt. Der Operator für Modulo in Java ist das Prozentzeichen „%“.

Wir können also folgendes Programm schreiben:




```
public class Operatoren {

    public static void main(String[] args) {
        int iWert = 13;
        int iDivisor = 5;
        System.out.println(iWert + " / " + iDivisor + " = " +
        (iWert/iDivisor) + " Rest " + (iWert%iDivisor));
    }
}
```

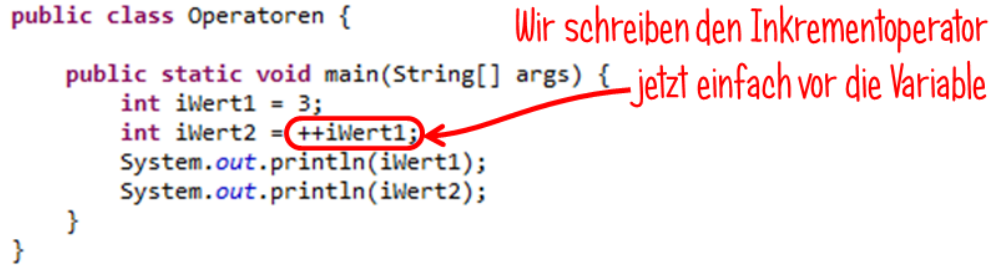
Sehen wir uns nun mal einen Operator an, der auch die Operanden verändert.




```
public class Operatoren {

    public static void main(String[] args) {
        int iWert1 = 3;
        int iWert2 = iWert1++;
        System.out.println(iWert1);
        System.out.println(iWert2);
    }
}
```

Dieser Code deklariert eine Variable iWert1 und weist den initialen Wert 3 zu. Anschließend weisen wir den Inhalt dieser Variable iWert2 zu, was wir auch tatsächlich bei der Ausgabe validieren können. Interessanterweise hat die Variable iWert1 aber bei der Ausgabe den Wert 4, also um eins erhöht. Dies macht der „++“ Operator, man nennt ihn auch Inkrementoperator. Er Erhöht also den Wert des Operanden um den Wert 1.

Wenn wir das mal ein bisschen sacken lassen, dann stellst sich früher oder später die Frage, warum die Java Autoren es so umgesetzt haben, dass zuerst die Zuweisung passiert und anschließend die Erhöhung. Andersrum wäre es vielleicht für manche Operationen sinnvoller gewesen. Sehen wir uns hierfür den folgenden Code an:




```
public class Operatoren {

    public static void main(String[] args) {
        int iWert1 = 3;
        int iWert2 = ++iWert1;
        System.out.println(iWert1);
        System.out.println(iWert2);
    }
}
```

Wenn wir den Code ausführen, sehen wir, dass beide Variablen die 4 aufweisen. In diesem Code wurde also zuerst die Variable iWert1 erhöht und anschließend der Wert zugewiesen.

Jetzt habe ich eine Fleißaufgabe für dich – überlege, wie der Dekrementoperator aussehen könnte (also Wert um 1 reduzieren)?? (Naja – ich wollte nur sehen, ob du noch konzentriert bist ☺)

Wir halten also fest:

- Eine Operation hat im Regelfall die Operanden, einen Operator und ein Ergebnis.
- Operatoren verändern oder verrechnen Werte.
- Klammern werden von innen nach außen verarbeitet.
- Es gibt Operatoren, welche die Operanden direkt verändern.
- Der Datentyp des Ergebnisses ist immer gleich dem größten Datentyp der Operanden.
- Inkrementoperatoren „++“ (und auch Dekrementoperatoren „--“) können vor und nach dem Operanden stehen.
- Steht der Inkrement/Dekrementoperator vor dem Operanden, wird die Operation zuerst ausgeführt und danach die Variable weiterverwendet (man spricht hier von pre-Inkrement bzw. pre-Dekrement).
- Steht der Inkrement/Dekrementoperator nach dem Operanden, wird zuerst die Variable für andere Operationen verwendet und anschließend die Inkrement/Dekrement Operation ausgeführt (man spricht hier von post-Inkrement bzw. post-Dekrement).

Die nächste Klasse von Operatoren sind die sogenannten „**relationalen Operatoren**“. Sie zeichnen sich dadurch aus, dass das Ergebnis vom Datentyp „boolean“ ist, also entweder „true“ oder „false“. Relationale Operatoren vergleichen immer zwei Werte und werden manchmal deshalb auch „Vergleichsoperatoren“ genannt. Da ich jedem zutraue die Operatoren zu verstehen, hier gleich mal alle in der Übersicht.

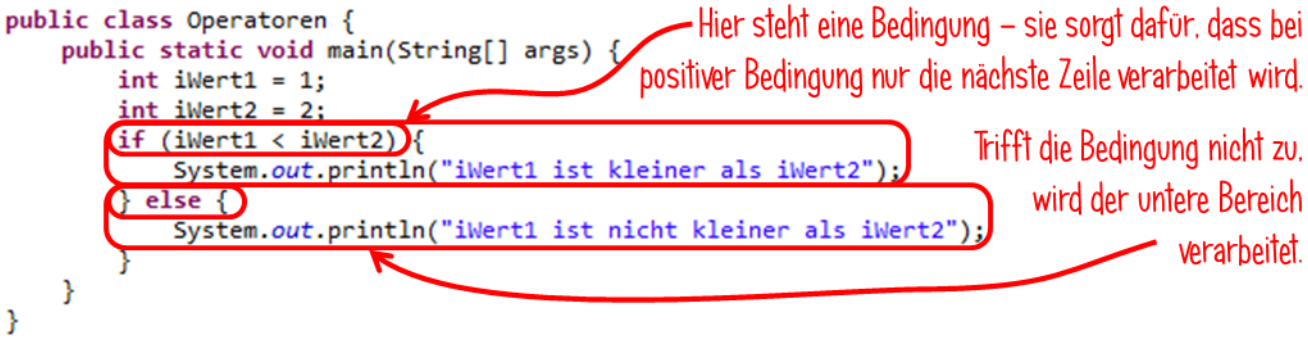
```
int iWert1 = 1;
int iWert2 = 2;
int iWert3 = 1;
```

Operator:	Symbol:	Befehl:	Ausgabe:
Gleich	==	`System.out.println(iWert1 == iWert2);`	false
		`System.out.println(iWert1 == iWert3);`	true
Ungleich	!=	`System.out.println(iWert1 == iWert2);`	true
		`System.out.println(iWert1 == iWert3);`	false
Größer	>	`System.out.println(iWert1 > iWert2);`	false
		`System.out.println(iWert2 > iWert1);`	true
Kleiner	<	`System.out.println(iWert1 < iWert2);`	true
		`System.out.println(iWert2 < iWert1);`	false
Größergleich	>=	`System.out.println(iWert1 >= iWert2);`	false
		`System.out.println(iWert2 >= iWert1);`	true
		`System.out.println(iWert1 >= iWert3);`	true
Kleinergleich	<=	`System.out.println(iWert1 <= iWert2);`	true
		`System.out.println(iWert2 <= iWert1);`	false
		`System.out.println(iWert1 <= iWert3);`	true

Eigentlich gibt es zu der Tabelle nichts weiter zu sagen – es ist relativ einfach. Eine Anmerkung kann ich mir aber nicht verkneifen – der Vergleich zweier Variablen mit einfachen Datentypen erfolgt mit **doppeltem Istgleich**! Ganze Armeen von Programmiernovizen versuchen es immer wieder mit einem einfachen Istgleich und laufen damit gegen die Wand (Java interpretiert ja das einfache Istgleich als Wertzuweisung – der Compiler würde also keinen Fehler melden). Also bitte immer darauf achten:

Wertvergleiche werden mit doppeltem Istgleich durchgeführt: ==

Die nächste Kategorie sind die **logischen Operatoren**. Diese dienen dazu, verschiedene boolean Werte zu verknüpfen. Hierzu müssen wir einen kleinen Vorgriff auf die Kontrollstrukturen machen – aber keine Angst, das Thema ist recht einfach zu verstehen. Sehen wir uns hierfür mal folgenden Code an:



Eine if-Anweisung ist also eine Möglichkeit, basierend auf einer Bedingung verschiedenen Code auszuführen. Ist die Bedingung „true“ also wahr, dann wird die nächstfolgende Zeile (bzw. der nächstfolgende Bereich in geschweiften Klammern) ausgeführt. Ist die Bedingung „false“ also falsch, dann wird die Zeile (bzw. der nächstfolgende Bereich in geschweiften Klammern) nach dem „else“ ausgeführt. In unserem Beispiel zeigt die Konsole folglich:

```
iWert1 ist kleiner als iWert2
```

Wofür brauchen wir den ganzen Kram nun eigentlich bei den logischen Operatoren?? Mit diesem Code können wir das Verhalten unserer Operatoren sehr einfach analysieren. Wir prüfen mal das Ergebnis von folgendem Code:

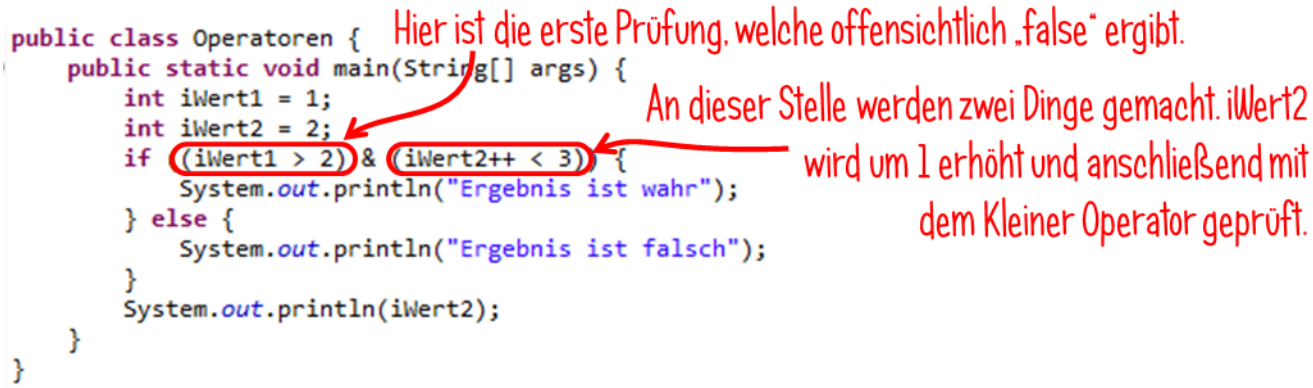
```
boolean bWert1 = true;
boolean bWert2 = false;
if (bWert1 & bWert2) {
      System.out.println("Ergebnis ist wahr");
} else {
      System.out.println("Ergebnis ist falsch");
}
```

Wenn Du den Code laufen lässt siehst Du, dass die Verzweigung in den „else“ Zweig läuft, also ist die Bedingung „false“. Und schwupps haben wir auch schon den ersten logischen Operator kennengelernt, die „UND“ Verknüpfung. Wie Du vielleicht schon weißt, ist eine logische UND Verknüpfung mittels einer Wahrheitstabelle darstellbar:

bWert1	bWert2	bWert1 & bWert2
false	false	false
false	true	false
true	false	false
true	true	true

Also, nur wenn beide Operatoren auf „true“ stehen, dann ist das Ergebnis ebenfalls „true“. In unserem Beispielcode war jedoch bWert2 auf „false“, ergo war das Ergebnis entsprechend der Tabelle „false“. Probiert ruhig mal mit dem Code rum und versucht die Tabelle zu verifizieren.

Wenn wir mal unsere Expertenbrille aufsetzen und in der Tabelle die ersten beiden Fälle betrachten, dann können wir doch bereits nach Prüfung des bWert1 Inhalts sagen, dass das Ergebnis der Verknüpfung „false“ sein muss, da bWert1 eben auf „false“ steht. Die große Frage ist nun, bricht Java nach der Analyse des ersten Operanden tatsächlich die Prüfung ab, oder prüft Java stur alle Operanden durch. Um dies festzustellen hilft uns folgender Code:



Wenn Java nun nach der ersten Prüfung, welche offensichtlich „false“ ergibt abbricht, dann kann Java die Variable iWert2 nicht um ein erhöhen. Wenn Java nicht abbricht, so müsste die letzte Ausgabe den Wert 3 sein, da iWert2 erhöht wurde. Wenn Du den Code ausführst siehst Du, dass tatsächlich die 3 da steht – Java also trotzdem es davon ausgehen muss, dass das Ergebnis der UND Verknüpfung auf jeden Fall „false“ ist, in die zweite Prüfung einsteigt.

Der Operator & hat jedoch noch einen Zwillingsbruder, den && Operator. Dieser ist auch eine UND Verknüpfung, bricht aber ab, sobald das Endergebnis feststeht. Probiert es aus – ändert im oberen Code den & Operator in einen && Operator und seht euch das Ergebnis an. Aus Performancegründen gilt:

Im Zweifelsfall ist der && Operator dem & Operator vorzuziehen.

Nun, wer sich schon ein wenig mit der boolschen Algebra beschäftigt hat weiß, dass es neben der „UND“ auch eine „ODER“ Verknüpfung gibt. Das Zeichen hierfür in Java ist das „|“ Symbol. Hier die Wahrheitstabelle:

bWert1	bWert2	bWert1 | bWert2
false	false	false
false	true	true
true	false	true
true	true	true

Wie ihr seht, ergibt die Verknüpfung immer dann true, wenn mindestens ein Operand „true“ aufweist. Auch hier sehen wir wieder eine Situation, bei der eine Prüfung bereits nach dem ersten Operanden abgebrochen werden kann. Wenn bWert1 auf „true“ steht, ist das Ergebnis zwingend „true“, egal welchen Wert bWert2 hat. Insofern sollte es uns nicht überraschen, wenn wir auch bei der „ODER“ Operation einen Zwillingsbruder haben für den gilt:

Im Zweifelsfall ist der || Operator dem | Operator vorzuziehen.

Jetzt fehlen noch zwei logische Operatoren. Beginen wir mit dem „Exklusiv Oder“, welcher mit dem Symbol „^“ in Java abgebildet wird. Hier die Wertetabelle:

bWert1	bWert2	bWert1 ^ bWert2
false	false	false
false	true	true
true	false	true
true	true	false

Wir sehen, dass das Exklusiv Oder immer dann „true“ ergibt, wenn exakt ein Operand auf „true“ steht und alle anderen auf „false“. Hier ist es demnach auch nicht sinnvoll einen „^^“ Operator zu definieren, da wir erst nach der Überprüfung des letzten Operanden sagen können, ob das Endergebnis „true“ oder „false“ ist.

Der letzte Operator ist der Negationsoperator. Dieser ist als „!“ definiert. Er dreht jedes „true“ in „false“ um und umgekehrt:

bWert1	!bWert1
true	false
false	true

Folgender Code wird also zu einer korrekten Ausgabe führen, da wir das Ergebnis des Vergleiches negieren:

```
int iWert1 = 1;
int iWert2 = 2;
if (!(iWert1 < iWert2)) {
      System.out.println("iWert1 ist nicht kleiner iWert2");
} else {
      System.out.println("iWert1 ist kleiner iWert2");
}
```

Die Konsole wird den Text „iWert1 ist kleiner iWert2“ zeigen.

Die letzten Operatoren, welche wir im Rahmen des Einsteigerkurses besprechen, sind die (arithmetischen) Zuweisungsoperatoren. Diese kümmern sich darum, dass Variablen mit neuen Werten belegt werden. Einen davon haben wir bereits kennengelernt – das einfache Istgleich „=“.
Mit diesem Operator in Kombination mit den arithmetischen Operatoren kann man alle anderen arithmetischen Zuweisungsoperatoren realisieren. Sehen wir uns folgenden Code an:

```
int iWert1 = 1;
int iWert2 = 1;
iWert1 += 3;
iWert2 = iWert2 + 3;
System.out.println(iWert1);
System.out.println(iWert2);
```

Wen der Code durchläuft siehst Du auf der Konsole zweimal den gleichen Wert, nämlich 4. Gehen wir die 4. Zeile mal schrittweise durch. Auf der linken Seite steht die Variable iWert2, in der das Operationsergebnis gespeichert werden soll. Auf der rechten Seite steht eine Addition. Nachdem zuerst das Ergebnis ermittelt wird und dieses im Anschluss in die Zielvariable gelegt wird, können wir die rechte Seite unabhängig von der Zielvariablen links interpretieren. Hier steht also: „Addiere zum Inhalt der Variable (welcher derzeit 1 ist) den Wert 3 hinzu“. Das Ergebnis wird wiederum in die Variable iWert2 geschrieben. Die Zeile ist also nicht mathematisch zu interpretieren (das wäre schließlich eine falsche Aussage), sondern prozedural.
Die Zeile darüber welche der Variablen iWert1 einen neuen Wert zuweist hat das gleiche Ergebnis aufzuweisen – insofern ist iWert += 3; lediglich eine Kurzsschreibweise für iWert = iWert + 3;

Diese Kurzschreibweise gibt es auch für andere Operationen:

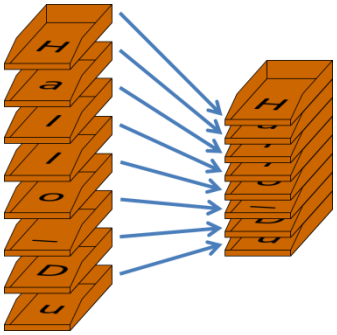
- Subtraktion (`iWert -=3;`)
- Multiplikation (`iWert *=3;`)
- Division (`iWert /=3;`)
- Modulo (`iWert %=3;`)

Es gibt noch weitere Operatoren – wie bspw. die bitweisen Operatoren, oder den Bedingungsoperator; für den Einsteigerkurs wollen wir es jedoch bei den Oben vorgestellten Operatoren belassen.

Wir halten also fest:

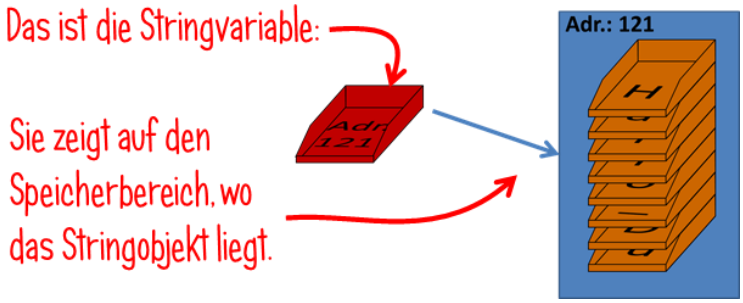
- Es gibt Vergleichsoperatoren, welche zwei Werte in Relation stellen. Das Ergebnis ist immer entweder „true“ oder „false“.
- Die Prüfung, ob der Inhalt zweier einfacher Datentypen identisch ist, erfolgt über den „==“ Operator, der von Programmieranfängern auch mal mit dem Zuweisungsoperator „=“ verwechselt wird.
- Logische Operatoren verknüpfen logische Werte („true“ oder „false“) miteinander. Hierbei gibt es „UND“, „ODER“, „EXCLUSIV ODER“ und die „NEGATION“
- Der „UND“ und „ODER“ Operator treten in zwei Formen auf. Bei der einfachen Notation „&“ bzw. „|“ wird jeder Operand auf jeden Fall berücksichtigt. Bei der doppelten Notation „&&“ bzw. „||“ wird die Prüfung der Operanden abgebrochen, sobald ein festes Endergebnis der Operation feststeht.
- Für die arithmetischen Operatoren gibt es auch Zuweisungsoperatoren, welche eine Kurzschreibweise darstellen. Hiermit kann bspw. für folgende Addition „iWert = iWert + 3“ in Kurzschreibweise folgendes geschrieben werden „iWert += 3“.

6 Strings – alles anders?



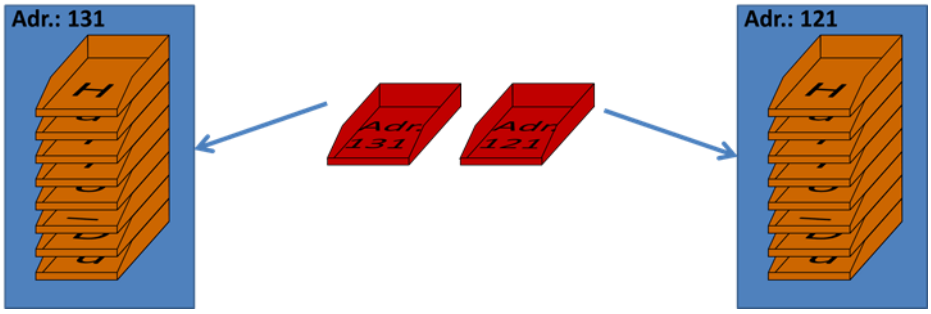
Prinzipiell haben wir alles Notwendige zum Thema Datentypen gelernt – wie man Variablen deklariert, wie man die Variablen initialisiert und wie man mit Hilfe von Operatoren die Werte anpasst. Wenn wir uns aber mal die Liste der Datentypen ansehen fehlt eine ganz wichtige – eigentlich der erste Datentyp, mit dem wir je zu tun hatten – der String (wir erinnern uns an unser „Hello World“ Programm). Als wir mit Datentypen begonnen hatten, wurde immer von einfachen Datentypen gesprochen. Diese haben sich dadurch ausgezeichnet, dass sie nicht aus anderen Datentypen zusammengesetzt werden. Strings sind aber Zeichenketten und somit eine Ansammlung von char Werten. Sie folgen somit eigenen Gesetzen. In diesem Kapitel werden wir uns also mal um diese, „neuen“ Gesetzmäßigkeiten kümmern.

Der erste Punkt, um den wir uns kümmern müssen ist die Struktur im Speicher. Wir haben ja gelernt, dass eine char Variable 2 Byte im Speicher beansprucht. Wenn wir nun den String „Hello, World!“ ablegen möchten, dann benötigen wir somit 13 char Werte. Nun könnten wir einfach sagen, dass eine Stringvariable schlichtweg 13 x 2 Byte in einer Folge im Speicher benötigt und die Stringvariable auf den ersten char – Datentyp zeigt, was das ‚H‘ wäre. Das würde vom Prinzip her funktionieren, hat aber den Nachteil, dass die Daten tatsächlich immer zusammenhängend im Speicher abgelegt werden müssen. Bei sehr großen Strings wird das zu einem Problem führen. Es gibt zwar Programmiersprachen, bei denen dies tatsächlich so realisiert wird, aber die Liste mitunter „fragmentiert“ abgelegt werden muss. Der String ist also eine „zusammenhängende Zeigerkette“. In Java ist dies aber etwas aufwändiger. Strings sind „Objekte“. Puh – schon wieder dieses Wort! Wir haben weiter Oben ja gesagt, dass ein Objekt eine Ansammlung von Daten (also unseren char Werten) und Funktionalitäten (also „Methoden“) ist. Das werden wir uns gleich näher ansehen. Doch davor müssen wir uns um die zentrale Frage kümmern, was ist eigentlich in der Variablen drin, welche vom Typ String ist? Der ganze String kann es nicht sein, weil er ja ein Objekt ist und somit auch Methoden innehat – ein String also einen ganzen Bereich im Speicher belegt.



In einer Stringvariablen finden wir immer eine Adresse – und zwar die des Stringobjektes. Das heißt irgendwo im Speicher residiert das Objekt und die Information wo im Speicher wir das Objekt finden, liegt in der eigentlichen Stringvariable. Insofern kann Java sich komplett autark um das Management des Speicherinhalts kümmern.

Das hat nun eine weitreichende Konsequenz! Wenn wir nun zwei Variablen mit dem gleichen Stringinhalt haben, dann haben wir ein ernsthaftes Problem zu prüfen, ob die beiden Strings gleich sind! Sehen wir uns folgendes Bild mal an:



Wir haben hier gleiche char – Sequenzen, aber unterschiedliche Objekte – dies erkennt man an den unterschiedlichen Adressen.

Wenn wir nun prüfen wollen, ob die beiden Variablen gleich sind gibt es zwei verschiedene Interpretationsweisen:

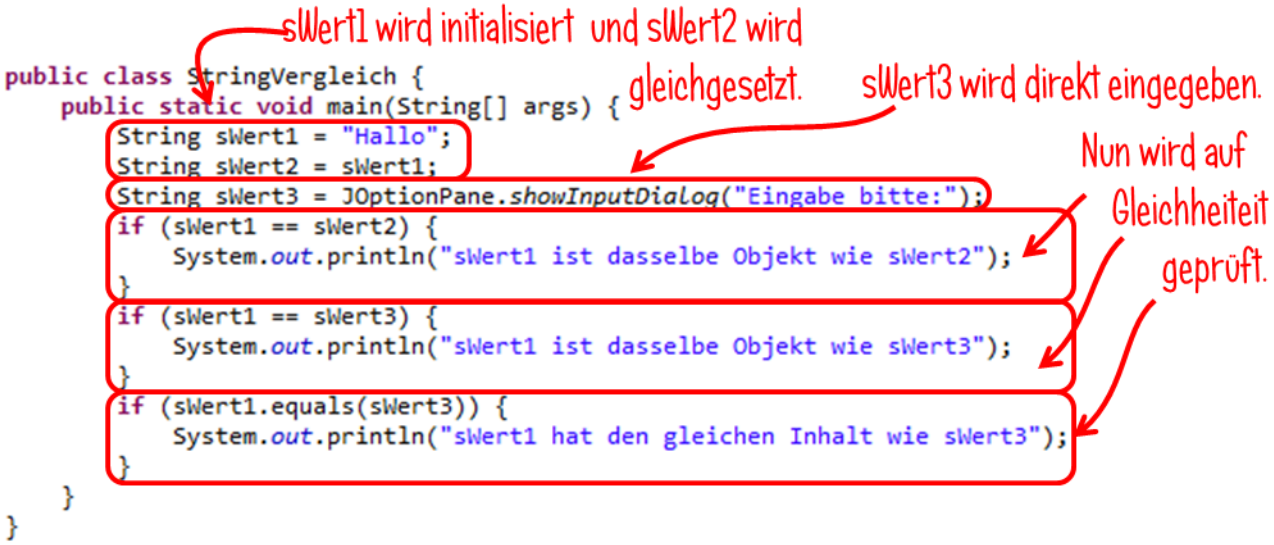
- Sprechen wir von den selben Objekten (also gleiche Adressen)?
- Sprechen wir von den gleichen Objekten (also gleiche Inhalte)?

Veranschaulichen können wir dies mit folgendem Beispiel: Wir haben zwei Autos, welche absolut baugleich sind. Sie haben aber unterschiedliche Nummernschilder. Insofern können wir davon ausgehen, dass es zwar die gleichen Autos sind, aber nicht dieselben Autos. Dies ist somit Vergleichbar mit der Situation, dass wir zwei Stringvariablen haben, welche zwar die gleichen Inhalte haben (also „baugleich“), jedoch nicht die gleichen Adresswerte (also andere „Nummernschilder“).

Wenn wir nun ein Bild von einem Auto sehen, können wir nun anhand des Nummernschildes feststellen, um welches Auto es sich exakt handelt. Wir vergleichen einfach die Nummernschilder und stellen fest, dass es sich bei dem Auto auf der rechten Seite um „Auto B“ handelt. „Mein Auto“ und „Auto B“ sind also dieselben Autos. Diese Situation versuchen wir nun mal mit einem kleinen Programm nachzustellen. Bevor wir dies tun müssen wir aber klären, wie wir prüfen ob zwei Objekte die gleichen Objekte sind und wie wir prüfen, ob zwei Objekte dieselben Objekte sind:

Prüfung, ob zwei Stringobjekte die gleichen sind:	**Prüfung, ob zwei Stringobjekte dieselben sind:**
`StringobjektA.equals(StringobjektB)`	`StringobjektA == StringobjektB`

Wie Du siehst, prüfe ich ob die Objekte dieselben sind über den eigentlichen Inhalt der Variable (was ja die Adresse ist) mit dem „==“ Operator. Das ist insofern auch logisch, da wir bei den einfachen Datentypen, welche ja als Inhalt den tatsächlichen Wert aufweisen auch diesen Operator für die Gleichheitsprüfung verwenden.




```
public class StringVergleich {
    public static void main(String[] args) {
        String sWert1 = "Hallo";
        String sWert2 = sWert1;
        String sWert3 = JOptionPane.showInputDialog("Eingabe bitte:");
        if (sWert1 == sWert2) {
            System.out.println("sWert1 ist dasselbe Objekt wie sWert2");
        }
        if (sWert1 == sWert3) {
            System.out.println("sWert1 ist dasselbe Objekt wie sWert3");
        }
        if (sWert1.equals(sWert3)) {
            System.out.println("sWert1 hat den gleichen Inhalt wie sWert3");
        }
    }
}
```

Wenn wir den Code laufen lassen und in die Eingabemaske „Hallo“ eintragen, dann sehen wir folgende Ausgaben:

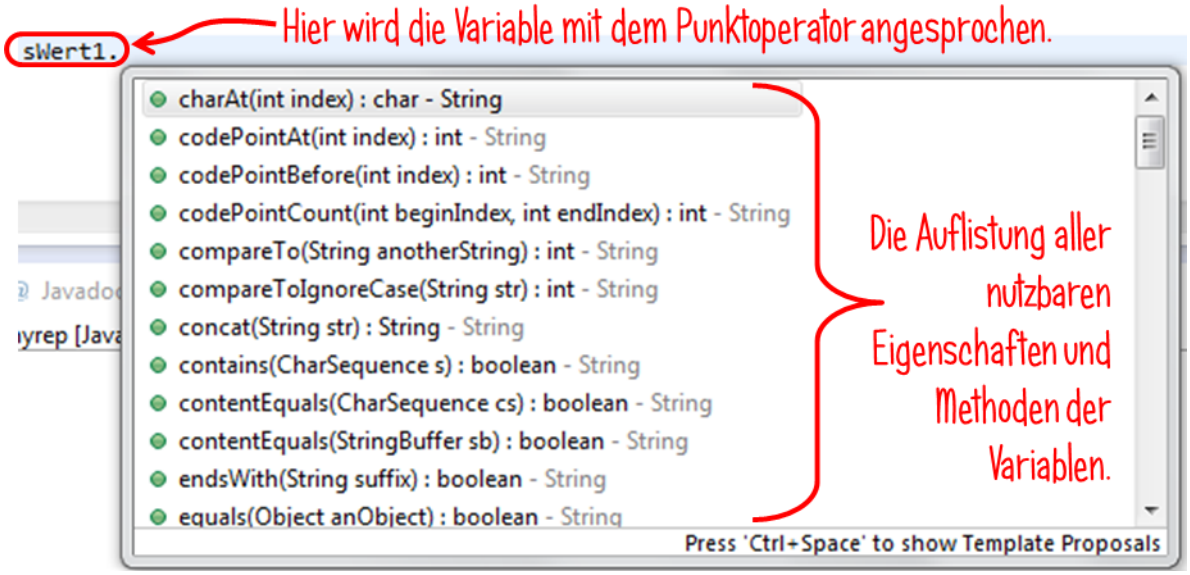
```
sWert1 ist dasselbe Objekt wie sWert2
sWert1 hat den gleichen Inhalt wie sWert3
```

Wir sehen, dass „sWert2“ nicht die gleiche Adresse hat wie „sWert3“, obwohl der Inhalt identisch ist. Die Inhaltsgleichheit stellen wir mit dem Befehl „equals“ fest. Er prüft also, ob der Inhalt von „sWert1“ und „sWert3“ identisch ist, obwohl es sich um unterschiedliche Objekte handelt.

Wir halten also fest:

- Bei Objekten steht in der eigentlichen Variable „lediglich“ eine Adressinformation, wo im Speicher sich die Daten und Methoden des Objektes befinden.
- Bei der Prüfung von „Gleichheit“ bei Objekten unterscheiden wir die Fälle, dass es sich um ein und dasselbe Objekt handelt oder ob es sich um zwei verschiedene Objekte mit dem gleichen Inhalt handelt.
- Bei Objekten prüfen wir mit dem „==“ Operator, ob es sich um ein und dasselbe Objekt handelt.
- Bei Strings prüfen wir mit der „.equals()“ Methode, ob der Inhalt von zwei Objekten identisch ist.

Nun wollen wir uns mal ansehen, welche Methoden so ein String noch mitbringt. Wer mit einem Programm wie Eclipse arbeitet, der hat vielleicht schon rausgefunden, dass wenn man eine Stringvariable hinschreibt und anschließend einen Punkt („.“) hinzufügt, dann erscheint ein Kontextmenü mit ganz vielen Einträgen. Hier noch ein kurzer Vorgriff auf das Thema „Objektorientierung“:
Ein Objekt hat ja Dateninhalte und Funktionalitäten („Methoden“ genannt). Auf diese Dinge muss man als Programmierer irgendwie zugreifen können. Dies erfolgt mit dem sogenannten Punktoperator. Alle für Außenstehende nutzbaren Eigenschaften und Methoden von Objekten können somit angesprochen werden. Eclipse weiß das und sucht automatisch die für uns nutzbaren Dinge heraus und verpackt sie in ein Kontextmenü.



Wie du siehst, gibt es jede Menge von Elementen in dieser Liste (scrollen nicht vergessen…). Wenn man ein erfahrener Programmierer ist, hat man schon einige dieser Methoden genutzt und weiß, was alles möglich ist – aber wie gesagt, das kommt nur mit der Erfahrung. Die Neulinge müssen sich immer mühsam durch die Liste arbeiten um zu sehen, welche Methode nun für die aktuelle Situation sinnvoll ist. Oder sie recherchieren dies im Internet (einfach in eine Suchmaschine „Java API String“ eingeben und du landest auf der richtigen Seite). Sehen wir uns mal ein paar wichtige Methoden von String an:

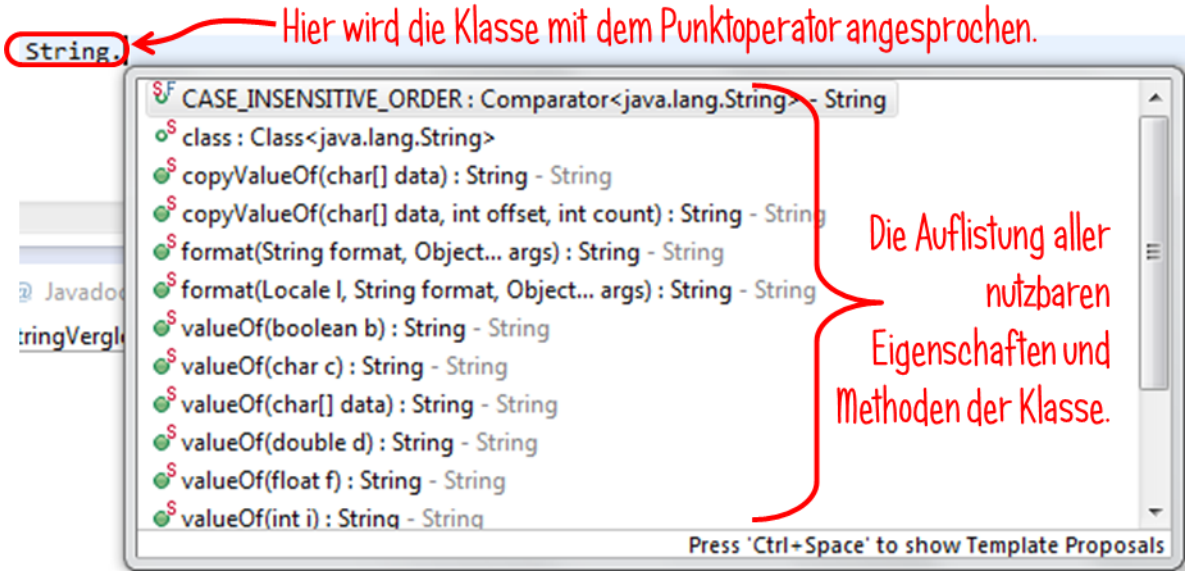
Methode	Erläuterung
`sWert1.equals("hallo");`	Prüft, ob der Inhalt des Arguments (also dem String der in den runden Klammern steht) identisch ist mit dem Inhalt von sWert1.
`sWert1.equalsIgnoreCase("hallo");`	Funktioniert wie „equals“, ignoriert aber die Grop/Kleinschreibung.
`sWert1.length();`	Gibt die Länge des Strings zurück (also Anzahl von char Werten).
`sWert1.charAt(0);`	Gibt eine char Variable zurück, welche den gleichen Wert hat wie das Zeichen an der angegebenen Stelle (hier die 0 -> also das allererste Zeichen).

Methode	Erläuterung
`sWert1.indexOf("a");`	Gibt die Fundstelle des Arguments in sWert1 an (hier also die Position, wo „a“ in dem Inhalt von sWert1 gefunden wird). Existiert der zu suchende String nicht, so wird -1 ausgegeben.
`sWert1.indexOf("a", 5);`	Funktioniert wie „indexOf“, jedoch veranlasst der zweite Parameter Java, erst ab der 5. Stelle zu suchen.
`sWert1.substring(1);`	Gibt einen Teilstring von sWert1 aus und zwar beginnend an der angegebenen Position bis zum Ende des Strings.
`sWert1.substring(1, 4);`	Gibt einen Teilstring von sWert1 aus und zwar beginnend an der angegebenen Position (1. Parameter) und endend vor der angegebenen Position (2. Parameter).

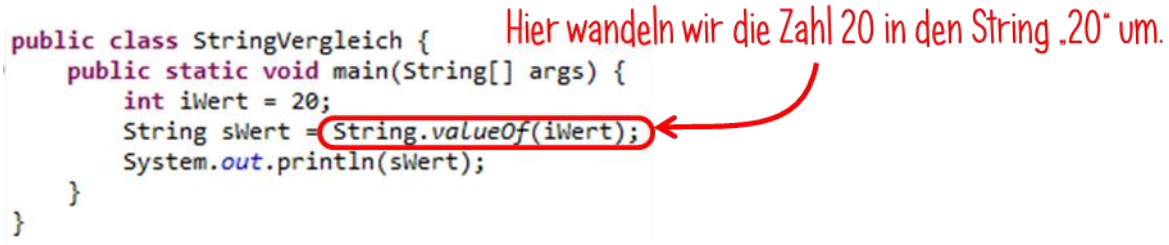
Das soll nur eine kleine Auflistung sein. Wenn Du mit den Funktionalitäten arbeiten wirst, dann erkennst Du schnell, was für die einzelne Situation sinnvoll nutzbar ist. Aber wie gesagt – das ist einfach nur „Übung“.

~~Blöde~~ Frage:
Wir haben ja viel über „typecast“ gesprochen. Kann ich irgendwie den Datentyp zwischen bspw. int und String wechseln?
Antwort: Nein und Ja. Nein, weil ein „typecast“ immer nur den Datentyp ändert, aber den eigentlichen Wert im Speicher nicht wirklich anfasst – es sei denn es wird etwas „abgeschnitten“. Werte ändern sich also nur aus der Konsequenz des Abschneidens. Ja, weil es trotzdem Methoden gibt, welche bspw. Strings interpretieren und Zahlen daraus generieren können bzw. weil es Methoden gibt, welche Zahlen in Stringformate umwandeln. Das ist aber kein „typecast“ mehr, sondern eine dedizierte Funktionalität. Wir werden gleich auf dieses Thema eingehen.

Beginnen wir mit der Richtung „einfachen Datentypen in einen String umwandeln“. Hier bietet die Stringklasse einige Funktionalitäten an. Die Methoden heißen „valueOf()“ – also „Wert von“. Sie wandeln einen einfachen Datentypen in einen String um. Dem aufmerksamen Leser schleicht sich jetzt vielleicht ein Stirnrunzeln ins Gesicht – da wir bis jetzt ja immer bereits einen String hatten, welcher eine Methode ausgeführt hat (bspw. hat sWert1.length() die Länge von sWert1 ausgegeben). Nun haben wir aber noch keinen String – wir wollen ihn ja erst erzeugen! Wie rufen wir also die Methode „valueOf()“ auf, wenn wir noch gar keinen String haben??
Um die Antwort zu verstehen müssen wie mal wieder in die Objektorientierung abtauchen. Und zwar bieten Klassen ihrerseits auch gewisse Funktionalitäten an – sogenannte „statische“ Methoden. Das sind Methoden die aufgerufen werden können, obwohl ich gar keine Variable von diesem Datentyp in der Hand halte. Um auf statische Methoden zugreifen zu können kann ich also nicht auf die Variablen den Punktoperator anwenden, sondern auf die Klasse. Da die Klasse „String“ heißt, probieren wir das gleich mal in Eclipse aus:



Und siehe da, hier tauchen auch die „valueOf“ Methoden auf. Für jeden einfachen Datentypen gibt es eine solche Methode. Probieren wir also folgenden Code mal aus:

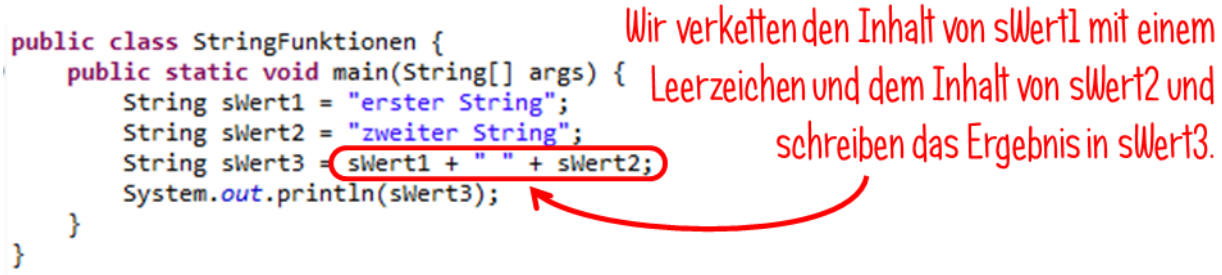


Wie du siehst, wird tatsächlich in den String der Wert „20“ geschrieben. Aber warum funktioniert das nicht einfach über einen Typecast. Sehen wir uns hierzu mal den Inhalt der Variablen iWert an:
`00000000 00000000 00000000 00010100`
Das ist einfach „nur“ die binäre Darstellung der Zahl 20.

Wenn wir nun einen Blick in die Stringvariablen sWert werfen, finden wir zwei char Variablen mit den folgenden Werten:
```
1. Variable (char '2'): 00000000 00110010
2. Variable (char '0'): 00000000 00110000
```

Es gibt keine Möglichkeit, ohne einen dedizierten Algorithmus zu bemühen, die beiden Werte ineinander zu überführen, indem man die Werte irgendwie gleichsetzt! Dieser Algorithmus zur Umwandlung von einfachen Datentypen in Strings steckt somit in der Methode „valueOf()“.

Es gibt allerdings noch eine zweite Möglichkeit, aus einem einfachen Datentypen einen String zu machen. Hierbei bedienen wir uns dem Stringoperator zum Verketten von Strings, der als Kürzel das „+“ Zeichen hat (ja, genau das gleiche Zeichen wie die arithmetische Addition, aber nachdem man Zeichenketten nicht wirklich arithmetisch addieren kann, wurde dieses Zeichen im Fall von Strings mit einer anderen Funktion belegt). Sehen wir uns mal den folgenden Code an:



Das Ergebnis auf der Konsole ist: „`erster String zweiter String`“. Nun haben die „Macher“ von Java diesen Verkettungsoperator recht schlau implementiert. Er erkennt, dass immer wenn ein + Operator in Verbindung mit mindestens einem String auftaucht, der Operator eine Stringverkettung durchführt und nicht addiert. Das heißt aber auch, dass wenn wir versuchen einen String mit einer Zahl mit Hilfe eines „+“ Operators zu verknüpfen, Java aus der Zahl erst mal einen String machen muss – was im Hintergrund nichts anderes Bedeutet als dass die „valueOf()“ Methode aufgerufen wird. Nun müssten wir eigentlich voraussehen können, was folgender Code auf die Konsole zaubert:

```
public class StringFunktionen {
	public static void main(String[] args) {
		String sWert1 = "die Zahl lautet";
		int iWert2 = 20;
		String sWert3 = sWert1 + " " + iWert2;
		System.out.println(sWert3);
	}
}
```

Wie Du wahrscheinlich schon erwartet hast, zeigt der Code „`die Zahl lautet 20`“ auf der Konsole an. Nun habe ich noch einen weiteren Code für Dich – versuche auf Basis des Oben erklärten Verhaltens vorher zu überlegen, was auf der Konsole stehen wird:

```
public class StringFunktionen {
      public static void main(String[] args) {
            String sWert3 = " lautet die Zahl.";
            int iWert1 = 3;
            int iWert2 = 4;
            String sWert4 = iWert1 + iWert2 + sWert3;
            System.out.println(sWert4);
      }
}
```

Wenn Du jetzt auf „34 lautet die Zahl.“ getippt hast, dann ist Dir eine Sache entgangen. Der Interpreter geht in der 6. Zeile rechts neben dem „=“ Zeichen Schritt für Schritt vor. Er sieht eine Integer Variable und danach ein „+“ Operator. Wenn nun der nächste Datentyp ein String wäre, dann würde der „+“ Operator eine Verkettung darstellen. Hier ist aber der nächste Datentyp ein Integer, also wird Java den „+“ Operator als Addition interpretieren. Es wird also der Wert von iWert1 und iWert2 addiert. Anschließend wird das Ergebnis mit dem nächsten Element verknüpft, was nun ein String ist. Also wird der „+“ Operator als Verkettung angesehen und das Ergebnis der Addition an den String sWert3 angehängt. Das Ergebnis ist somit „`7 lautet die Zahl.`“.

~~Blöde~~ Frage:
Kann ich die Verarbeitungsreihenfolge irgendwie steuern?
Antwort: In gewissen Grenzen geht das – und zwar mit Klammern. Diese werden immer von innen nach außen verarbeitet und bieten Dir somit eine Möglichkeit, das Verhalten zu kontrollieren.

Wir halten also fest:

- Der „+“ Operator ist bei Strings ein Verkettungsoperator – er verkettet verschiedene Strings miteinander.
- Sollte bei der Verkettung eine der beiden Operanden ein einfacher Datentyp sein, dann wird dieser implizit umgewandelt.
- Die Umwandlung kann auch erzwungen werden, indem man die „valueOf()“ Methode verwendet. Diese gibt das Argument der Methode, welches immer ein einfacher Datentyp ist als Stringrepräsentation zurück.
- Die Operationsverarbeitung läuft von links nach rechts. Wenn jedoch von links gesehen zwei Zahlenvariablen stehen, so wird der „+“ Operator als Addition ausgeführt.
- Mit Hilfe von Klammern können wir die Operationshierarchie verändern, da Klammern immer von innen nach außen verarbeitet werden.

Jetzt kommt noch der letzte zu beachtende Punkt bei Strings. Ein String kann ja jede beliebige Zeichenkette beinhalten. Eine valide Zeichenkette ist aber auch der Leerstring – das ist sozusagen eine Zeichenkette ohne Zeichen (jaja, was es alles gibt…). Beim Programmieren würde ein Leerstring als "" eingegeben werden. Ein Leerstring ist also ein valider String! Im Gegensatz zum Wert null (also nicht der String "null" sondern der Wert null). Dies ist kein valider String, sondern in den meisten Programmiersprachen das Schlüsselwort für „Nichts“. Das ist also eine Stringvariable, welche keinen String beinhaltet. Den Wert null können übrigens nur Objekte annehmen, nicht aber einfache Datentypen.

Zusatzinfo: Das liegt daran, dass Objektvariablen in Java lediglich eine Adressinformation beinhalten, welche angibt, wo die Stringdaten im Speicher sind. Liegen in der Objektvariablen keine Adressinformationen vor, so ist der String null.

So, wer das jetzt geschafft hat, darf sich jetzt hochoffiziell einen Kaffee gönnen und mal durchschnaufen. Wenn der Groschen noch nicht gefallen ist, dann einfach nochmal in Ruhe den Text durchgehen und Punkt für Punkt die Informationen sacken lassen. Es hört sich wirklich schlimmer an, als es ist. Immer daran denken –

der Rechner arbeitet einfach nur stur seine Regeln ab. Wer diese verstanden hat, dem kann der Rechner auch nichts mehr vormachen. Versprochen!

7 Wrapperklassen – der Zahlenwerkzeugkasten

Java zeichnet sich ja dadurch aus, dass es eine objektorientierte Programmiersprache ist. Alle Funktionalitäten sind somit in Klassen und Objekten verpackt. Das macht die Sache für den Einsteiger nicht wirklich einfacher – der Fortgeschrittene behauptet aber immer steif und fest, dass er nie anders programmieren möchte, als objektorientiert. Nun – da wir in diese Sphären noch nicht aufgestiegen sind, müssen wir den Experten erst mal glauben und hoffen, dass sie Recht behalten. Bis dahin müssen wir uns einfach daran gewöhnen, dass wir irgendwie mit Objekten und Klassen arbeiten, ohne die Details zu verstehen. Aber das kommt alles noch… Das Problem ist nun aber, dass wir in einem vorausgegangenen Kapitel von „einfachen Datentypen“ gesprochen haben. Das sind nun wirklich keine Objekte. Was machen wir aber, wenn wir Funktionen benötigen, die uns das Leben leichter machen sollen. Folgendes Beispiel soll uns eine solche Situation mal näher bringen: Wir haben ja die Methode „valueOf()“ bei den Strings kennen gelernt, bei der ein einfacher Datentyp in eine Stringrepräsentation umgewandelt wird. Versuchen wir uns mal an dem umgekehrten Weg – wir wollen eine Stringdarstellung einer Zahl in eine tatsächliche Zahl umwandeln. Verwenden wir hierfür wieder die gleichen Werte wie im Stringbeispiel:

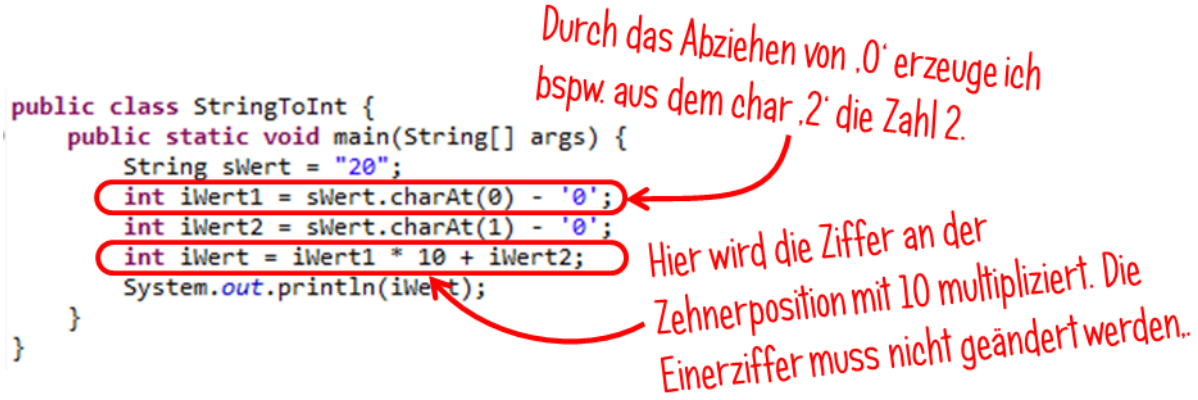
Der String soll den Wert „20“ haben, was aus folgenden beiden Characterwerten zusammengesetzt ist:

```
1. Wert (char '2'): 00000000 00110010
2. Wert (char '0'): 00000000 00110000
```

Ziel ist eine Integervariable mit dem Zahlenwert 20:

```
00000000 00000000 00000000 00010100
```

Folgendes kleine Programm soll uns dies nun ermöglichen (da wir noch keine Schleifen kennen habe ich das Programm sehr einfach und unflexibel zusammengestrickt):

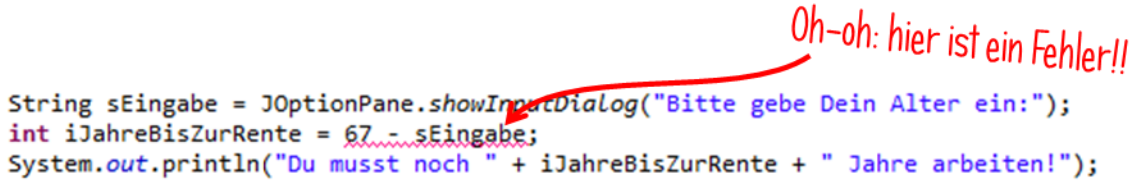



```
public class StringToInt {
    public static void main(String[] args) {
        String sWert = "20";
        int iWert1 = sWert.charAt(0) - '0';
        int iWert2 = sWert.charAt(1) - '0';
        int iWert = iWert1 * 10 + iWert2;
        System.out.println(iWert);
    }
}
```

Dieser Code wandelt mir also eine zweistellige Zahl in Stringdarstellung in eine tatsächliche Zahl in eine Integervariable um. Er holt sich den Characterwert an der Stelle 0 und zieht die Characterrepräsentation ‚0‘ ab – was nichts anderes ist als die Zahl 48 (wir erinnern uns an die ASCII Codes). Dadurch wird aus der ‚0‘ die Zahl 0, aus ‚1‘ die Zahl 1 usw. Gleiches gilt für die Position 1. Beim Rechnen wird die vordere Ziffer mit 10 multipliziert, was bei einer zweistelligen Zahl hoffentlich jedem einleuchtet. Zum Schluss werden die Zahlen noch addiert und der Wert ist fertig berechnet.

Dieser Code ist zugegebenermaßen sehr unflexibel und wird niemanden so richtig glücklich machen. Vielleicht frägt sich jetzt der ein- oder andere nun, „Ja und? Wer braucht das schon?“. Nun hier muss ich klar sagen – wir alle werden früher oder später eine Funktion brauchen, bei der wir aus einem String eine Zahl machen.

Sehen wir uns folgenden Code an:



Tja, so kann es nicht gehen. Wir können ja keinen String von einer Zahl abziehen. Also müssen wir hier eine Umwandlung machen. Unser kleines Umwandlungsprogramm von Oben könnte uns hier helfen, wenngleich es recht fehleranfällig ist. Die gute Nachricht ist, dass Java uns eine schöne Funktion anbietet, welche genau solche Umwandlungen durchführt. Schreibe den Code mal ab und ersetze die mittlere Zeile durch folgende:

```
int iJahreBisZurRente = 67 - Integer.parseInt(sEingabe);
```

Wie Du siehst, ist der Fehler nun verschwunden und wenn ihr das Programm ausführt seht ihr, dass Java uns tatsächlich unsere verbleibende Arbeitszeit in Jahren ausgibt (sofern die Politik das Alter nicht noch weiter anhebt…). Was aber ist nun dieses „Integer“ – ist das das Gleiche wie int? Die Antwort ist „nein, aber…“. Integer ist eine sogenannte „Wrapperklasse“ für int. Hinter „int“ steht der einfache Datentyp und hinter „Integer“ die Wrapperklasse für „int“. Sie sind also sehr eng verwandt. Die Wrapperklasse beinhaltet so einige nützliche Funktionen in Zusammenhang mit int Werten, auf die ich hier nur im Ansatz eingehen werde – weitere Informationen bekommst Du, wenn Du in Eclipse einfach nur „Integer.“ eingibst und das sich darauf öffnende Menü durchscrollst. Trotzdem hier die wichtigsten Methoden der Integer Klasse:

Methode:	**Bedeutung:**
Integer.parseInt(str)	Der String str wird nach einer Integer Zeichenkette geparst und als int Datentwert zurückgegeben.
Integer.decode(str)	Dekodiert einen String, der als: Hexwert (0x, 0X oder # mit folgender Hexzahl, bspw. ergibt 0XA0 den Wert `160`) oder Oktalwert (0 mit folgender Oktalzahl bspw. ergibt 010 den Wert 8) in eine Dezimalzahl.
Integer.toHexString(i)	Der Dezimalwert i wird in eine Hexadezimalzahl in Stringrepräsentation gewandelt.
Integer.toOctalString(i)	Der Dezimalwert i wird in eine Oktalzahl in Stringrepräsentation gewandelt.
Integer.toBinaryString(i)	Der Dezimalwert i wird in eine Binärzahl in Stringrepräsentation gewandelt.
Integer.toString(i)	Der Dezimalwert i wird in einen String umgewandelt.

Alle anderen Funktionalitäten sind eher exotisch, können aber bei machen speziellen Problemen echt viel Arbeit ersparen. Also, einfach mal durch die angebotenen Methoden in Eclipse stöbern.

Natürlich haben die anderen einfachen Datentypen auch Wrapperklassen. Insofern ist es für uns nun kein Problem zu interpretieren, was die folgenden Methoden so machen:

```
Double.parseDouble("100.012");
Float.parseFloat("1.2");
Long.parseLong("12345");
Boolean.parseBoolean("true");
```

Bei der Boolean Wrapperklasse kann man sich das Parsen auch sparen, indem man folgenden Code eingibt:

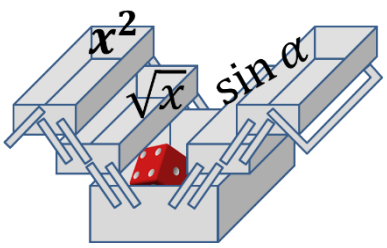
```
sValue.equals("true")
```

Das Parsen ist hier also einfach nur eine Prüfung auf „true“oder „false“. Im Gegensatz zu den Zahlenparse - Funktionen würde das parsen nach Boolean auch keine Fehlermeldung liefern, wenn als Argument ein Wert übergeben wird, der kein boolean ist (bspw. der Wert „SoTrue!!“). Vermutlich ist die Java interne Umsetzung ähnlich aufgebaut, wie unser kleine Alternativmethode. Wichtig ist jedoch noch, dass bei den Zahlenparse-funktionen ein Wert, der nicht interpretiert werden kann einen Fehler wirft, was das ganze Programm anhält. Wie man damit umgeht werden wir später lernen, wenn wir uns mit try/catch Blöcken auseinandersetzen. Probier das aber ruhig mal aus – trage in die Methode „parseInt“ mal einen Buchstaben ein und sehe, was passiert…
Weiterhin noch der Hinweis, dass die Wrapperklassen auch die Grenzwerte der einzelnen einfachen Datentypen beinhalten. So liefert bspw. Integer.MAX_VALUE den Wert `2147483647`.

~~Blöde~~ Frage:
Ich sehe alle möglichen einfachen Datentypen, außer char. Gibt es denn keinen Character.parseChar()?
Antwort: Nein. Es gibt zwar eine Wrapperklasse „Character“, diese ist aber im Regelfall für ganz spezielle char – Operationen bzw. char – Analysen gedacht. Das Parsen nach einen Character würde jedoch auch nicht wirklich sinnvoll sein, da ein String ja ohnehin nur aus char Werten besteht (bzw. null oder ein Leerstring ist). Die Stringmethode „charAt()“ ist hier zu nutzen, wenn ich einen einzelnen Characterwert aus einem String herauslösen möchte.

Wir halten also fest:

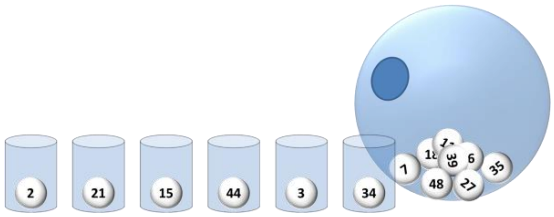
- Zu jedem einfachen Datentypen gibt es eine Wrapperklasse.
- In der Wrapperklasse sind Hilfsfunktionalitäten, welche uns den Umgang mit den einfachen Datentypen vereinfachen sollen.
- Allen voran sind die „parse“ Funktionen, welche aus einer Stringrepräsentation eines Wertes des einfachen Datentyps, diese in einen entsprechenden Datenwert umwandeln (bspw. aus dem String „20“ wird durch Integer.parseInt(„20“) der Wert 20).
- Die Wrapperklassen beinhalten auch Informationen über den einfachen Datentyp, wie bspw. Maximal- und Minimalwerte.
- Bei char müssen wir nicht parsen, da ein String ohnehin aus char Werten besteht.



Aber das ist noch lange nicht alles. Um wirklich mit Zahlen arbeiten zu können müssen wir mit ihnen rechnen. Gut, die einfachen Operatoren haben wir kennen gelernt – aber wie sieht es mit komplexeren Rechnungen wie Wurzel ziehen oder Quadrieren aus? Hierfür bietet uns Java die „Math“ Klasse. Das ist eine Sammlung von nützlichen Mathematikfunktionen, welche uns das Leben noch leichter machen werden. Auch bei der Math Klasse gilt, einfach mal in Eclipse „Math.“ eingeben und das Kontextmenü durchblättern.

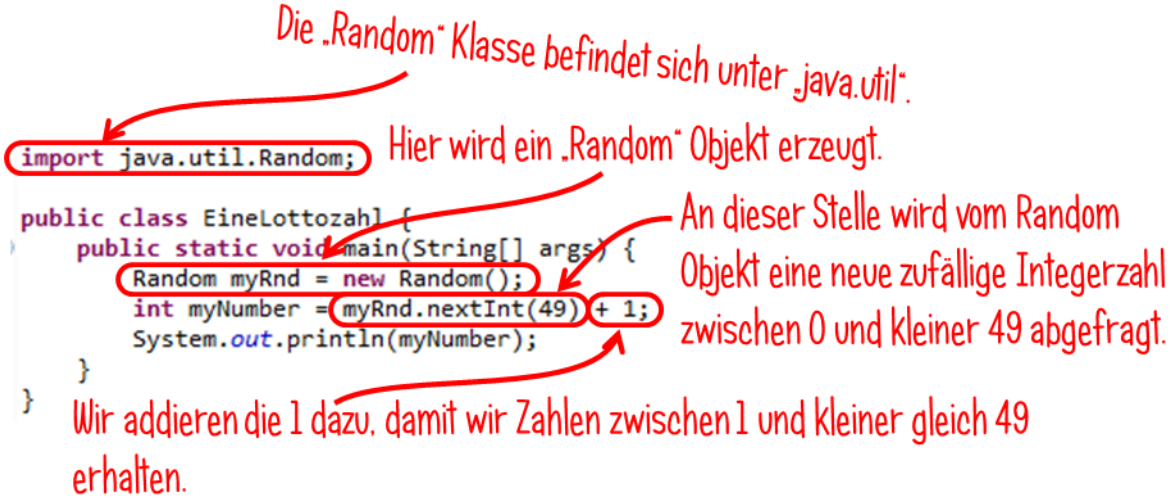
Hier eine kleine Aufstellung der wichtigsten Methoden.

Methode:	**Bedeutung:**
Math.sqrt(dVal)	Die Methode gibt die Quadratwurzel der double - Zahl „dVal“ zurück. Das Ergebnis ist ebenfalls vom Typ „double“.
Math.pow(dBasis, dExp)	„pow“ steht für Power – also potenzieren. Hierbei wird der Parameter dBasis hoch dExp gerechnet: $dBasis^{dExp}$. Sowohl die Parameter, als auch das Ergebnis ist vom Typ „double“.
Math.sin(a) Math.cos(a) Math.tan(a)	Dies sind die trigonometrischen Funktionen, wobei die Winkel in Radiant angegeben werden (also nicht 180° sondern π). Auch hier sind alle Werte vom Typ „double“.
Math.PI	Die Zahl als π Doublewert (3.141592653589793)
Math.random()	Eine double Zufahllszahl zwischen 0.0 und 1.0



Zufallszahlen sind bei Spiel-Programmen oftmals wichtig. Die Zufallszahl der „Math“ Klasse ist etwas störrisch zu handhaben. Insofern bietet Java gleich eine eigene Klasse für Zufallszahlen, die – wer hätte das gedacht – Random heißt.

Folgendes Programm soll die Handhabung der Randomklasse verdeutlichen:



Wichtig zu verstehen ist, dass es beim Computer eigentlich keine Zufallszahlen gibt. Vielmehr wird ein Algorithmus angewendet, der eine pseudozufällige Sequenz erzeugt. Das ist im Prinzip eine Rechnung, welche immer wieder auf das Vorergebnis angewendet wird und scheinbar immer eine neue zufällige Zahl ausspuckt. Daher ist es auch wichtig, dass der Initialwert (also der Startwert der Rechnung) zufällig gesetzt wird. Wenn wir das Random Objekt mit „new Random()“ erzeugen, dann wird die aktuelle Systemzeit in Millisekunden verwendet. Man nennt dies „Seeddata“ – also die „Saat“ der Zufallszahl. Ist diese nicht zufällig, so wird die Zufallszahl auch nicht zufällig sein.
Um dies zu verdeutlichen können wir die Seeddata auch vorgeben und sehen, was passiert. Probier mal folgenden Code aus, bei dem wir die Zufallszahl mit einem festen Seedwert „100“ erzeugen:

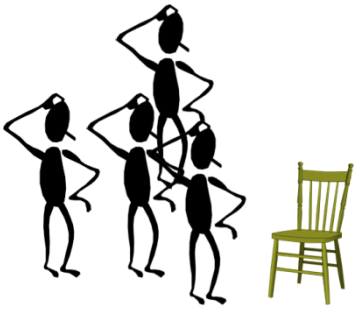
```
Random myRnd = new Random(100);
System.out.println(myRnd.nextInt(49) + 1);
System.out.println(myRnd.nextInt(49) + 1);
System.out.println(myRnd.nextInt(49) + 1);
```

Wie Du siehst, gibt Java drei Zufallszahlen aus. Wenn Du aber das Programm nochmal laufen lässt, dann stehen wieder exakt die gleichen drei Zufallszahlen da! Erst wenn Du die 100 aus dem Parameter in der ersten Zeile entfernst, dann kommen immer neue Zufallszahlen heraus.

~~Blöde~~ Frage:
Warum kann man überhaupt einen Seedwert setzen? Die Systemzeit sollte doch für uns ausreichen, oder?
Antwort: Prinzipiell ist das richtig. Wenn man es aber ganz genau nimmt, ist die Systemzeit auch nicht wirklich zufällig. Es gibt Systeme, bei denen der User eine zufällige Bewegung mit der Maus macht, welche dann mit Hilfe eines eigenen Algorithmus in eine long – Zahl umgewandelt und als Seeddata verwendet werden kann.

8 Arrays

Du hast nun gelernt, wie man einfache Informationen in Variablen abspeichert und wie man sie wieder rausbekommt, bzw. wie man damit arbeitet. Hierbei haben wir einfache Datentypen wie bspw. int für ganze Zahlen oder double für Gleitkommazahlen kennen gelernt. Aber auch Zeichenketten als zusammengesetzte Datentypen in den String Objekten haben wir genutzt. Soweit ist das eine recht praktische Sache. Nun gibt es aber leider immer wieder Situationen, bei denen dies nicht ausreichend ist. Was machen wir, wenn wir mehrere gleichartige Daten irgendwo abspeichern wollen. Gut – wir könnten einfach mehrere Variablen deklarieren und die Werte dort ablegen. Das geht auch bis zu einem gewissen Grad, aber wenn es mal mehr wie fünf Variablen sind, dann wird es schon recht mühsam den Code für fünf Variablen zu kopieren. Hier müssen wir irgendwie etwas Einfacheres finden. Stellen wir uns doch nur mal vor, wir müssten von einem Reisebus alle Namen der Mitreisenden abspeichern. So ein Bus fasst gut und gerne 60 Personen. Wer Lust hat 60 Variablen einzeln zu deklarieren und zu verarbeiten, der sollte an dieser Stelle aufhören zu lesen, denn das Kapitel ist dafür gedacht, solche unsinnigen Arbeiten zu vermeiden. Diese Leute müssen ohnehin mit ihrer Zeit haushalten ☺



Doch im Ernst – wir sollten uns beim Programmieren immer angewöhnen, Redundanzen zu vermeiden. Ein Codefragment, welches mehr als einmal im Programm vorkommt deutet im Regelfall auf ein nicht effizient gestaltetes Programm hin.

Die Lösung für ein derartiges Problem heißt „Array“. Arrays gibt es in fast jeder Programmiersprache und ist im Grunde nichts anderes als eine Liste von Variablen gleichen Datentyps. In dieser Liste können wir nun mehrere Werte gleichzeitig ablegen. Hierbei gibt es zwei Einschränkungen (zumindest in den meisten Programmiersprachen und somit auch bei Java): Wir müssen **vorher wissen, wie viele Elemente** in dem Array gespeichert werden sollen und wir können immer nur Daten des **gleichen Datentyps** ablegen.

~~Blöde~~ Frage:
Warum soll ich mir Gedanken über die Anzahl der Werte machen, wenn ich einfach ein Array mit – sagen wir mal der Größe 10000 erzeugen kann? Das reicht mit Sicherheit!
Antwort: Das ist zwar eine Möglichkeit welche funktioniert, sie ist aber von der Speicherausnutzung alles andere als effizient. Wir verdammen unsere User bei jedem Aufruf des Programms unnötig viel Speicher zu verbrauchen und das nur, weil wir während des Programmierens zu faul waren nachzudenken! Also im Zweifelsfall lieber mal in die Details gehen und ein Array mit einer angepassten Größe erstellen.

Nun, wie können wir uns ein solches Array nun vorstellen. Es ist im Prinzip vergleichbar mit einer Liste von Daten. Sehen wir uns mal folgenden Zettel des letzten Sportturniers an. Wir haben verschiedene Namen aufgeschrieben, wobei wir die Positionen von oben nach unten durchnummerieren. Wenn wir nun den Namen einer Platzierung haben wollen, dann sehen wir nach der Platznummer und können daneben den Namen lesen. Nachdem wir ja die Nummern absteigend notiert haben, finden wir den Namen auch sehr schnell. Andersrum ist es da schon etwas schwieriger. Wenn wir wissen wollen auf welchem Platz die Hanna Zamers ist, dann müssen wir die ganze Liste durchsuchen, bis wir fündig werden.

Sehen wir uns mal an, wie eine solche Liste in einem Array aussehen würde:

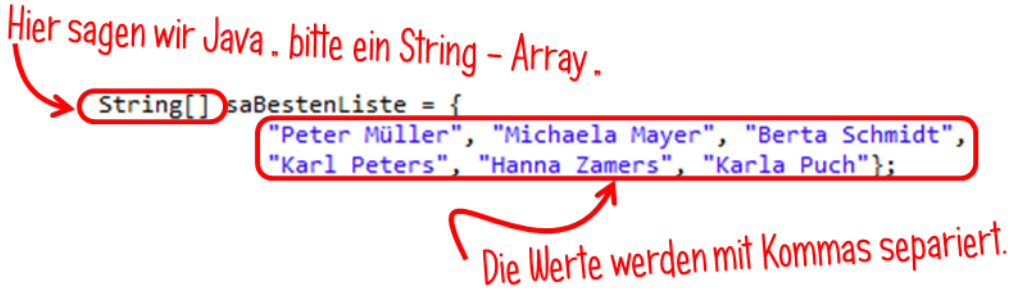
```
String[] saBestenListe = {
      "Peter Müller", "Michaela Mayer", "Berta Schmidt",
      "Karl Peters", "Hanna Zamers", "Karla Puch"};
```

Wie Du siehst, gibt es keine Benennung der Position der einzelnen Werte. Dies geschieht automatisch – sie werden von links nach rechts durchnummeriert. Aber Achtung! Der allererste Eintrag bekommt die Indexposition 0! Insofern ist der letzte Wert auf der Position 5 und nicht 6… Die Liste sieht im Rechner also wie rechts dargestellt aus.

Index:	Wert:
0	Peter Müller
1	Michaela Mayer
2	Berta Schmidt
3	Karl Peters
4	Hanna Zamers
5	Karla Puch

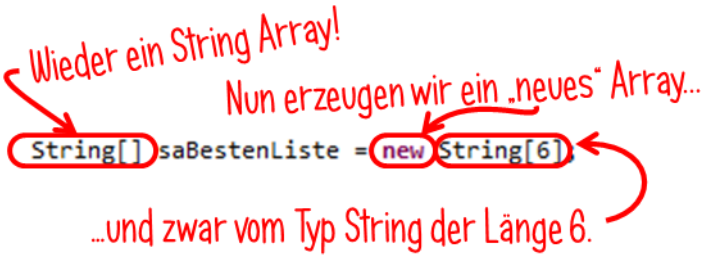
Über diese Situation stolpern die meisten Neulinge. Ein Array mit 6 Einträgen hat die Indexwerte 0 bis 5. Übrigens der Grund für die Nutzung der 0 in der Informatik ist, dass die 0 eine valide Adresse ist, die wir nicht „wegschmeißen“, sondern mitnutzen sollen.

Aber sehen wir uns den Syntax für die Arraydeklaration nochmal an. Das vorher gemachte Beispiel ist eine Möglichkeit, das Array bei der Erzeugung bereits mit Initialen Werte zu füllen. Wir werden im Anschluss noch sehen, wie wir Arrays ohne initaler Bewirtschaftung erzeugen.



Die beiden eckigen Klammern nach dem Datentypen der Deklaration besagen, dass ein Array des entsprechenden Datentypen erzeugt werden soll. Danach kommt wie üblich der Variablenname und die Zuweisung erfolgt über eine geschweifte Klammer, in der kommasepariert die Datenwerte – hier eben Strings – stehen.

Bis auf die Tatsache, dass man die Schreibkonvention lernen muss, ist das doch gar nicht so schwer oder? Sehen wir uns mal eine andere Art der Deklaration an:

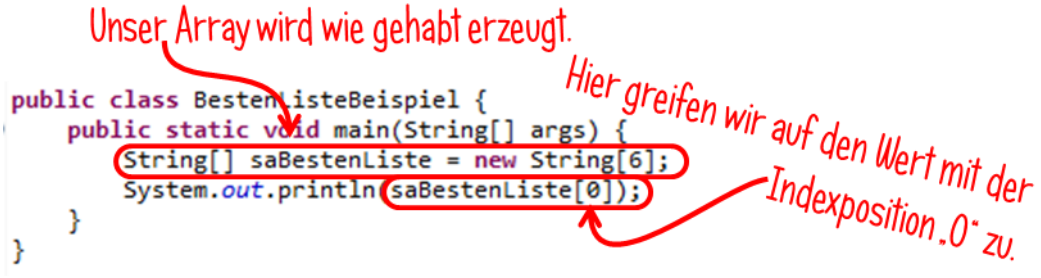


Was jetzt neu ist, ist die rechte Seite der Zuweisung. Wir schreiben hier „new“ um ein neues Objekt zu erzeugen, gefolgt von String[6], was so viel heißt wie „Stringarray der Länge 6“. Der Wert 6 steht im oberen Code explizit – also als Konstante – da. Es kann aber auch eine Integervariable sein, welche die Länge des Arrays als Wert mitbringen soll.

~~Blöde~~ Frage:
Wieso muss ich das Schlüsselwort String zweimal angeben? Es sollte doch eigentlich reichen, wenn man String[] nur am Anfang schreibt oder?
Antwort: Zugegeben, momentan drängt sich der Gedanke auf. Wenn das Thema „Objektorientierung“ mal angesprochen wird, dann werden wir sehen, dass man Objekte auch in Variablen schreiben kann, welche von einem anderen, übergeordneten Datentyp sind. Da das für uns momentan aber noch nicht relevant ist, müssen wir uns jetzt einfach nur merken, dass wir das Schlüsselwort String bei der Variablendeklaration (also vorne) und bei der Objekterzeugung (also hinten, nach dem „new“) benötigen.

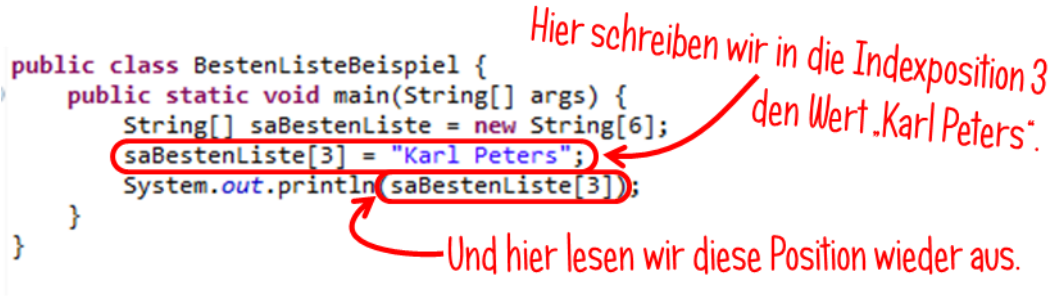
Jetzt haben wir also ein String-Array der Länge 6 erzeugt, also ein Gebilde, welches 6 Stringwerte aufnehmen kann. Die erste Frage, die sich stellt ist, was ist denn nun initial in den 6 Arrayfeldern drin? Schließlich haben wir ja nur gesagt, dass wir ein Array der Länge 6 brauchen und nicht, wie im vorausgegangenen Beispiel, was

darin sein soll. Um das nun rauszufinden, müssen wir nun die Werte des Stringarrays ausgeben. Dies geht mit folgendem Programm:



Wenn wir den Code nun laufen lassen, dann zeigt uns die Konsole den Wert null an. Das heißt also, dass zwar ein Array erzeugt wurde, die einzelnen Stringelemente dieses Arrays aber keinen Inhalt aufweisen. Das ist durchaus sinnvoll, da Java sich nicht erdreistet vorauszusehen, was wir eigentlich im Array haben wollen. Darum müssen wir uns wohl oder übel selbst kümmern.

Sehen wir uns hierfür kurz den Zugriff auf Arrays nochmal an. Wie wir im vorausgegangenen Code gesehen haben, müssen wir für den Zugriff nach dem Variablennamen einfach eine eckige Klammer setzen und darin die Indexposition. Das funktioniert übrigens beim Lesen genauso wie beim Schreiben:



Wie Du siehst, gehen wir mit Arrayvariablen genauso um, wie mit einfachen Variablen, nur dass wir die Indexposition mit angeben müssen. Ansonsten hat sich nichts geändert.

So – nun mal eine gute Nachricht: Jeder Datentyp, egal ob einfach oder zusammengesetzt kann als Array deklariert werden. Wenn wir bspw. ein Array der Länge 10 mit ganzen Zahlen haben wollen, schreiben wir einfach folgenden Code:

```
int[] iaMyArray = new int[10];
```

Wer bis jetzt sehr aufmerksam war müsste sich nun die Frage stellen, welche Initialwerte bei einem solchen Array nun eingetragen werden – null geht ja bei einfachen Datentypen nicht. Es muss also irgendein anderer Wert sein. Und wie finden wir das raus? Genau! Wir erzeugen das Array und geben uns eine Position aus. Wenn wir das ausprobieren, dann sehen wir, dass int Arrays (genauso wie alle anderen Zahlendatentypen) die 0 als Initialwert haben.

Anmerkung: Bei boolean Arrays ist übrigens „false“ der Initialwert.

~~Blöde~~ Frage:
Wer hindert mich eigentlich daran bei einem Array der Größe 6 auf, sagen wir mal auf die Position 8 zu schreiben?
Antwort: Der Javainterpeter wird Dich daran hindern. Wenn Du auf eine nicht existierende Indexposition zugreifst, dann wird ein Fehler erzeugt (IndexOutOfBounds Exception) und das Programm wird angehalten. Der User würde vom „Absturz“ sprechen. Also beim Arrayzugriff immer aufpassen!

Um zu verhindern, dass man fehlerhaft zugreift ist es nun möglich, die Länge eines Arrays festzustellen. Hierzu gibt es die .length Eigenschaft. Folgender Code würde somit die Länge des Arrays auf der Konsole ausgeben:

```
System.out.println(saBestenListe.length);
```

Wir werden später bei den Kontrollstrukturen sehen, wie man mit einer Schleife und dieser length Eigenschaft das gesamte Array ausliest oder beschreibt.

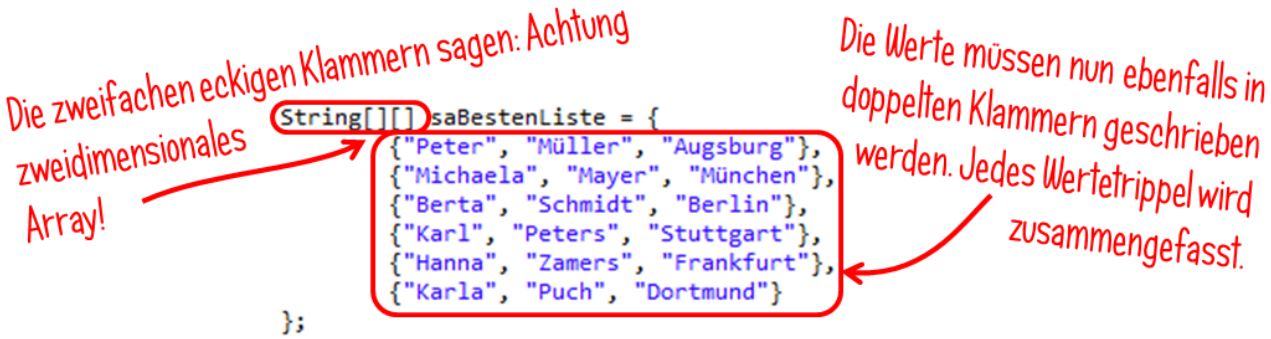
Soweit – so gut. Wir haben nun verstanden, wie wir Arrays erzeugen – entweder mit von uns vorgegebenen Initialwerten, oder als leeres Array mit von Java vorgegebenen Initialwerten. Wir wissen, wie man die Länge eines Arrays feststellt und wie wir Daten rein- und wieder raus kriegen. Wir haben alles was wir brauchen, oder?

Nicht ganz! Zwei Dinge muss ich noch erklären – dann können uns die Arrays keine Angst mehr machen.

Beginnen wir mit den mehrdimensionalen Arrays. Wie der Name schon sagt, können wir Arrays mehrdimensional auslegen. Wenn wir in einem Array bspw. die Vornamen, Nachnamen und den Wohnort eintragen müssen, dann reicht uns das normale Array einfach nicht mehr aus. Dann sollte es wie rechts dargestellt erzeugt werden.

		Vorname:	Nachname:	Wohnort:
		Spaltenindex		
		0	1	2
Zeilenindex	0	Peter	Müller	Augsburg
	1	Michaela	Mayer	München
	2	Berta	Schmidt	Berlin
	3	Karl	Peters	Stuttgart
	4	Hanna	Zamers	Frankfurt
	5	Karla	Puch	Dortmund

Wie Du siehst, gibt es einen Zeilenindex und einen Spaltenindex. Beide sind notwendig, um eine einzelnen Position in dem Array zu identifizieren. Der Nachname „Peters“ ist beispielsweise in Spalte 1 und Zeile 3 zu finden. Die Arraydeklaration für ein solches Konstrukt sieht wie folgt aus:



Soweit, so gut. Anhand der doppelten eckigen Klammern sagen wir Java, dass wir ein zweidimensionales Array speichern wollen. Bei der initialen Belegung der Felder werden diese nun nicht mehr als einzelne Werte in einer geschweiften Klammer eingetragen, sondern als in geschweiften Klammern zusammengesetzte Werte. Ein zweidimensionales Array ist also nichts anderes, als ein eindimensionales Array, in dem wiederum eindimensionale Arrays liegen. In manchen Programmiersprachen (bspw. Javascript) wird das auch tatsächlich so angelegt. In Java wird es in der obenstehenden Kurzform angelegt, wobei auch eine explizite Erzeugung pro Zeile möglich ist, wodurch wir sogar unterschiedliche Größen pro Zeile haben können. Für’s Erste soll aber ein „rechteckiges“ Array – also eines bei dem jede Zeile die gleiche Größe hat – reichen.

Diese Oben beschriebene Form wird allerdings selten genutzt (lediglich für konstante Arrays). Meistens werden die Arrays leer angelegt und anschließend dynamisch mit Werten belegt. Der Folgende Code zeigt auf, wie ein leeres zweidimensionales Array erzeugt wird:

```
String[] saBestenListe = new String[6][3];
```

~~Blöde~~ Frage:
Ist es eigentlich egal, ob ich new String[6][3] oder new String[3][6] erzeuge?
Antwort: Sehr gute Frage! Vom Speicherplatz her gesehen ist es tatsächlich egal, ob das Array 6x3 oder 3x6 Felder groß ist. Insofern wäre die Antwort „ja“. Vom Handling her gesehen ist es überhaupt nicht egal, da wir im Regelfall pro Zeile zusammenhängende Werte haben möchten und nicht pro Spalte. Sehen wir uns das im Folgenden nochmal genauer an…

Wir haben ja die .length Eigenschaft kennengelernt. Wir lassen mal folgenden Code laufen:

```
String[][] saBestenListe = new String[6][3];
System.out.println(saBestenListe.length);
```

Java gibt uns den Wert „6“ aus. Das heißt also, dass uns Java die Anzahl der Zeilen angibt. Hängen wir nun den unten stehenden Code dran:

```
System.out.println(saBestenListe[0].length);
```

Nun erhalten wir die Ausgabe „3“ – also die Anzahl der Spalten oder besser gesagt die Anzahl der Datensätze in der nullten Zeile. Das hat nun für das Handling eine wichtige Konsequenz. Wenn wir zusammenhängende Datensätze ablegen (wie bei uns Vorname, Nachname, Wohnort), dann sollten wir die einzelnen Datensätze pro Zeile zusammenfassen und nicht pro Spalte. Insofern ist es am sinnvollsten das Array 6x3 Felder groß zu machen und nicht 3x6 Felder. Das erleichtert später die Interpretation der Inhalte ungemein. Glaub‘ mir das – ich weiß aus bitterer Erfahrung, dass ein willkürlich zusammengewürfeltes mehrdimensionales Array viele unnütze Abendstunden vor dem Rechner bescheren kann!

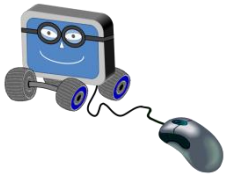
Preisfrage – wie kann ich nun ein fünfdimensionales Array erzeugen? Genau! Man macht einfach fünf Klammern hintereinander. Also – wie Du siehst ist wieder alles ganz streng logisch!

Wir halten also fest:

- Arrays sind eine feste Zusammenstellung (oder Liste) von mehreren Werten gleichen Datentyps.
- In Java können Arrays, welche einmal erzeugt wurden, von der Länge her nicht mehr verändert werden.
- Arrays können für jeden beliebigen Datentyp erstellt werden. Hierbei wird bei der Deklaration dem Datentyp einfach ein Paar eckiger Klammern nachgestellt („[ ]“).
- Arrays mit von uns definierten initialen Werten werden bei der Deklaration bewirtschaftet, indem die Werte kommasepariert in geschweiften Klammern zugewiesen werden.
- Arrays ohne von uns definierten initialen Werten werden mit dem Schlüsselwort „new“ und einem darauf folgendem Datentyp mit eckigen Klammern initialisiert. In den eckigen Klammern schreiben wir die Arraygröße rein.
- Arrays können auch mehrdimensional sein. Hierbei werden einfach mehrere eckige Klammern hintereinander geschrieben.

~~Blöde~~ Frage:
Ich finde es sehr unbefriedigend, dass man bei Arrays vorher schon wissen muss, wie groß sie werden. Gibt es da nicht eine Alternative?
Antwort: Ja, die gibt es. Sollten wir eine Liste von Werten dynamisch erweitern müssen, dann bietet sich die sogenannte ArrayList an. Diese werden wir aber zu einem späteren Zeitpunkt besprechen. Für’s erste soll das normale Array erst mal reichen.

9 Alles unter Kontrolle



Da sind wir ja schon ganz schön weit gekommen. Wir wissen nun richtig viel über Variablen, Operatoren, ja sogar Arrays. Das Problem ist aber, dass wir mit diesem Wissen eigentlich noch gar nicht richtig programmieren können. Streng genommen sind wir lediglich in der Lage Daten im Rechner zu speichern, diese irgendwie zu verknüpfen und anschließend wieder auszugeben. Das ist noch nicht wirklich das Gelbe vom Ei.

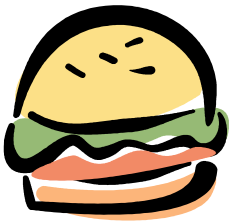
Wir müssen also irgendwie Aktionen abhängig von gewissen Gegebenheiten durchführen können. Bspw. müssen wir bei Fehleingaben den User nochmals bitten, diese zu wiederholen. Wir brauchen also die Kontrolle über das Programm. Die Elemente, welche uns diese Kontrolle ermöglichen nennen wir daher auch „**Kontrollstrukturen**".

Wenn wir das ganze Thema von Grund auf angehen – und das tun wir ja immer oder? – dann sehen wir, dass wir die einfachste Form der Kontrollstruktur bereits genutzt haben: die **Sequenz.**

„Sequenz" ist eigentlich nur ein wichtig klingender Ausdruck für die Anweisung an den Computer: „Führe alle folgenden Anweisungen der Reihe nach aus!". Zugegebenermaßen ist der Begriff „Sequenz" nicht nur kürzer, er hört sich eben auch toller an. In all unseren bisherigen Programmen, in denen wir mehr als einen Befehl eingegeben haben, wurde eine Sequenz programmiert. Ohne dass Du es wusstest! Wer hätte das gedacht?!?

Dem aber nicht genug. Um Programmalgorithmen unabhängig von Programmiersprachen dokumentieren zu können, haben zwei Informatiker namens Isaac Nassi und Ben Shneiderman eine Diagrammform entwickelt, welche folgerichtig auch „Nassi Shneiderman Diagramm" heißt und eben auch eine Sequenz darstellen kann. Man nennt diese Diagramme übrigens auch **Struktogramm** – nicht weil sie so strukturiert sind (das ist unser Code auch), sondern weil sie strukturiertes Arbeiten fördert. Das Struktogramm für ein Programm mit drei Anweisungen sieht wie rechts dargestellt aus.

Anweisung 1
Anweisung 2
Anweisung 3



Wie Du (hoffentlich) aus dem Bild interpretieren kannst, werden die Anweisungen von Oben nach Unten abgearbeitet. Weiterhin ist zu bemerken, dass die Anweisungen nicht einer Programmiersprache entsprechen müssen – es sind frei formulierte, interpetationsfreie Anweisungen, welche von einem Programmierer verstanden werden können. Mit solchen Struktogrammen kann man alle möglichen Prozesse darstellen – bspw. wie man vom Auto aus im Drive In Schalter an einen Burger kommt.

Gebe Bestellung ab
Fahre zum Kassenschalter
Zahle den Preis
Fahre zum Ausgabeschalter
Nehme Burger entgegen

Wir fahren an die Gegensprechanlage und bestellen den Burger. Danach fahren wir vor und bezahlen und am letzten Schalter wartet der Burger auf uns. Ein Schritt nach dem anderen. Wenn wir am letzten Element angekommen sind, dann ist der Prozess (oder das Programm) der Bestellung beendet. Das Essen ist dann wiederum ein eigener Prozess – wobei man bei manchen Zeitgenossen gerade im Schnellrestaurant hier ohnehin nicht von „Essen" reden kann…

Wer um ca. 12:45 mal an einem vielbesuchten Drive In Schalter etwas Exotisches bestellt, der muss durchaus schon mal einen Zwischenschritt einlegen – und zwar auf dem Warteparkplatz. Hier haben wir nun genau den Fall, dass wir flexibel auf eine bestimmte Situation reagieren müssen. Insofern reicht unsere Sequenz als einziges Kontrollelement nicht mehr aus. Wir brauchen eine Abfrage. Hier kommt die **Verzweigung** ins Spiel. Wir haben sie ja bereits kurz bei den logischen Operatoren kennengelernt.

Eine Verzweigung ist ein Konstrukt, welches zuerst eine Frage stellt, welche mit „**Ja**“ oder „**Nein**“ beantwortbar ist und je nachdem wie die Antwort lautet, entweder der „Ja“ Zweig oder der „Nein“ Zweig durchlaufen wird. Sehen wir unseren erweiterten Bestellvorgang mal an.

Zuerst fällt mal auf, dass wir ein neues Element haben, die Verzweigung. In dem Dreieck der Verzweigung steht die vorhin erwähnte Frage, welche immer nur mit Ja oder Nein zu beantworten ist. Je nachdem, welche Situation vorherrscht, wird nun der linke oder rechte Weg eingeschlagen. In der rechts dargestellten Situation besteht der linke und rechte Zweig jeweils nur aus einer Anweisung. Dies muss nicht so sein – hier können beliebig viele, zur Not auch ineinander verschachtelte Struktogrammelemente stehen.

Wenn nun der eingeschlagene Zweig abgearbeitet wurde, dann geht die Verarbeitung irgendwann wieder in den ursprünglichen Hauptpfad über – sofern nach der Verzweigung das Programm nicht ohnehin beendet ist. Wie sieht das nun im Java Code aus? Nachdem ein automatischer Burgerbedienroboter für uns noch eine zu große Programmierherausforderung ist, versuchen wir uns in einem andern Programm. Wer in den oberen Kapiteln aufgepasst hat – und das gilt ja für uns alle…. – wird folgendes Programm noch im Kopf haben:

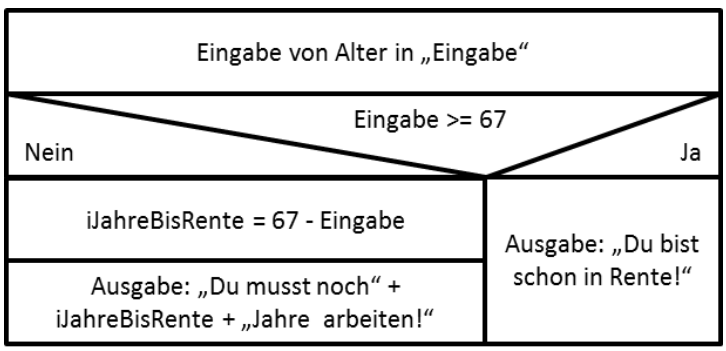
```
import javax.swing.JOptionPane;
public class RentenTest {
    public static void main(String[] args) {
        String sEingabe = JOptionPane.showInputDialog("Bitte gebe Dein Alter ein:");
        int iJahreBisZurRente = 67 - Integer.parseInt(sEingabe);
        System.out.println("Du hast noch " + iJahreBisZurRente + " zu arbeiten!");
    }
}
```

Wir haben hier ja ausgerechnet, wie lange wir noch bis zur Rente arbeiten müssen. Es gibt hier aber ein kleines Problem! Was ist, wenn Dein Opa sein Alter angibt? Angenommen er wäre 78 Jahre alt, dann würde das Programm folgende Ausgabe erzeugen:

```
Du musst noch -11 Jahre arbeiten!
```

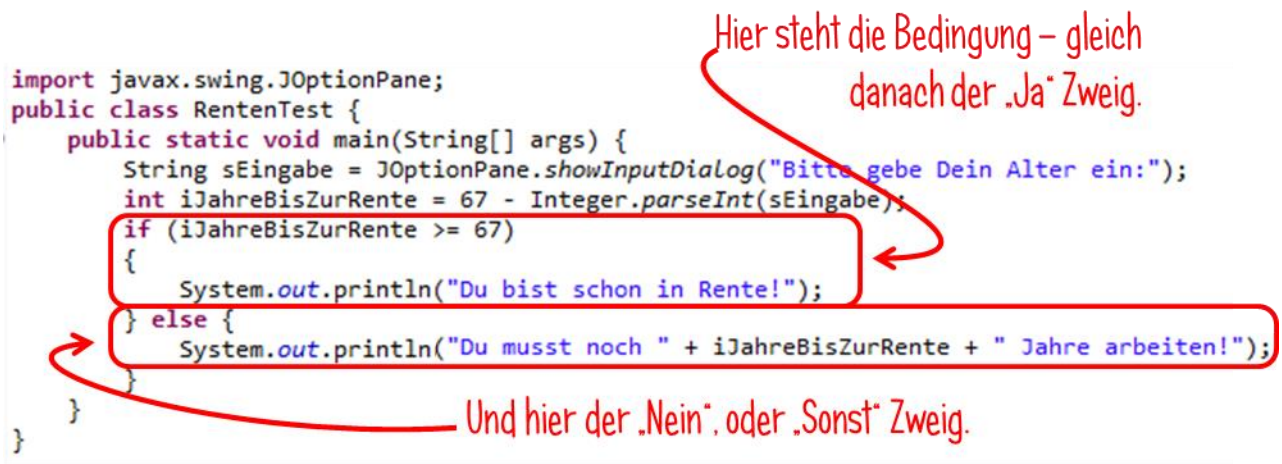
Nun, hier würde Dein Opa aber Augen machen (vor allem, wenn er aufgrund Altersweitsichtigkeit das Minus übersieht)! Das ist also ein klarer Fall für eine Verzweigung.

Sehen wir uns mal hierzu mal ein Struktogramm an, welches dieses Problem umgehen kann:



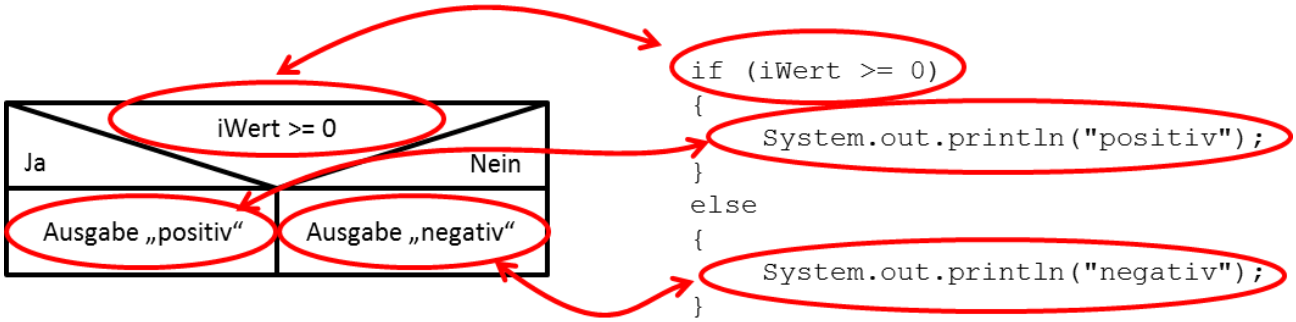
Diesen Algorithmus zu interpretieren dürfte nun für dich kein Problem sein.

Sehen wir uns den dazugehörigen Java Code mal an:




```
import javax.swing.JOptionPane;
public class RentenTest {
    public static void main(String[] args) {
        String sEingabe = JOptionPane.showInputDialog("Bitte gebe Dein Alter ein:");
        int iJahreBisZurRente = 67 - Integer.parseInt(sEingabe);
        if (iJahreBisZurRente >= 67)
        {
            System.out.println("Du bist schon in Rente!");
        } else {
            System.out.println("Du musst noch " + iJahreBisZurRente + " Jahre arbeiten!");
        }
    }
}
```

Wer genau hinsieht, erkennt auch das Java Konstrukt für eine Verzweigung. Das Schlüsselwort hierfür ist „if", gefolgt von der Bedingung in runden Klammern. Der „Nein" oder besser „Sonst" Zweig folgt nach dem Schlüsselwort „else". Die beiden Zweige werden durch geschweifte Klammern begrenzt. Hier nochmal ein Struktogramm und ein Java Code in der Gegenüberstellung:




```
if (iWert >= 0)
{
    System.out.println("positiv");
}
else
{
    System.out.println("negativ");
}
```

Dieser Syntax findet sich so oder in ähnlichen Formen in vielen anderen Programmiersprachen wieder.

~~Blöde~~ Frage:
Ich habe schon mal ein Java Programm gesehen, bei dem nach der schließenden runden Klammer der „if" Anweisung keine geschweifte Klammer gekommen ist, sondern nur ein einzelner Befehl. Ist das ein fehlerhaftes Programm gewesen?
Antwort: Nein, es ist wahrscheinlich nicht fehlerhaft gewesen, wenngleich ich es als „unschön" bezeichnen würde. Der Hintergrund ist, dass bei einer „if" Anweisung der „Ja" Zweig im Prinzip nur der nächste Befehl ist – also bspw. eine println Anweisung. Wenn aber eine geschweifte Klammer folgt, dann ist der nächste Befehl einfach nur eine Zusammenfassung von mehreren weiteren Befehlen, wodurch wir einen „Ja" Zweig erhalten, welcher eben mehrere Befehle beinhaltet. Es zeugt aber von einem guten Programmierstil, wenn man immer die geschweiften Klammern nutzt – auch wenn man nur einen einzigen Befehl im „Ja" Zweig benötigt.

Gehen wir nun noch etwas tiefer in die Verzweigung hinein. Ich hatte weiter oben ja gesagt, dass die Bedingung der Verzweigung (also das, was innerhalb der runden Klammern steht), nur „Ja" oder „Nein" sein darf. Also bspw. iWert >= 0 ist entweder zutreffend, also „Ja" oder nicht zutreffend und somit „Nein". Leider muss ich zugeben, dass das für Java nicht 100% korrekt formuliert ist. Wenn ich es ganz genau nehme, sind die beiden Zustände nicht „Ja" und „Nein", sondern „true" und „false". Jetzt mag der ein oder andere vielleicht denken „Ja und? Ist das wirklich so wichtig?". Die Antwort ist – „Ja! Das ist wichtig!". Die beiden Werte „true" und „false" haben wir ja bereits kennen gelernt und zwar bei den boolean Variablen. Diese können ja auch nur „true" und „false" annehmen.
Mit anderen Worten – die Bedingung einer Verzeigung ist vom Datentyp „boolean". Hört sich jetzt zwar ein bisschen blöd an – ist aber so! Auch eine Bedingung hat einen Datentyp.

Das hat nun zur Folge, dass folgender Code tatsächlich funktioniert:

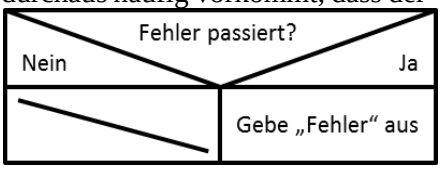
```
int iWert = 3;
boolean bWertIstPositiv = iWert >= 0;
if (bWertIstPositiv)
{
      System.out.println("Der Wert ist positiv!");
}
```

Die Variable „bWertIstPositiv“ ist vom Typ boolean und somit qualifiziert sie sich als eine valide Bedingung für eine Verzweigung. Das bringt nun eine ganze Fülle von Anwendungsmöglichkeiten mit sich. Wenn wir bspw. eine Variable namens „bEinFehlerIstPassiert“ haben, dann können wir im Programm diese auf „true“ setzen, sobald tatsächlich irgendein Fehler aufgetreten ist. Und im Code würde man folgenden Abschnitt sehr einfach verstehen:

```
if (bEinFehlerIstPassiert)
{
      System.out.println("Fehler!");
}
```

Also – mit einer schlauen Struktur wird auch plötzlich unser Code sehr einfach lesbar!

Bevor ich es vergesse – wenn Du Dir den oben stehenden Code nochmal ansiehst merkst Du vielleicht, dass ich den kompletten „else“ Zweig weggelassen habe. Das ist erlaubt, da es durchaus häufig vorkommt, dass der „Nein“ oder „Sonst“ Zweig nicht benötigt wird. Man spricht hier von einer „einseitigen Verzweigung“, auch wenn das beim genaueren Hinsehen ein eher merkwürdiger Begriff ist. Aber, so ist das nun mal! Das Struktogramm sieht aber wie gehabt aus, lediglich der „Nein“ Zweig bleibt leer, bzw. manche Autoren kennzeichnen ein leeres Element auch mit einem Querstrich.



Einen weiteren Punkt muss ich an dieser Stelle auch noch ansprechen. Wir haben bei den Operatoren die sogenannten „logischen Operatoren“ angesprochen und für diverse Tests ja die Verzweigung schon kurz eingeführt. Vor dem Hintergrund des Datentyps der Bedingung dürfte nun auch klar sein, dass wir beliebig viele logische Operationen innerhalb der Bedingung verschachteln können – solange das Endergebnis ein boolean Datentyp bleibt – was bei logischen Verknüpfungen immer der Fall ist. Also – bevor wir viele Verzweigungen ineinander verschachteln sollten wir prüfen, ob wir das Ganze nicht einfacher umsetzbar ist. Vergleiche einfach mal die beiden Codefragmente:

```
if (iWert > -10)
{
    if (iWert < 10)
    {
        System.out.println("Zahl ist einstellig");
    }
}
```

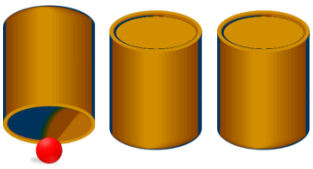
```
if ((iWert > -10) && (iWert < 10))
{
    System.out.println("Zahl ist einstellig");
}
```

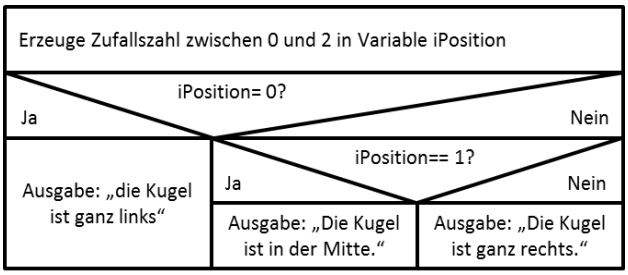
Irgendwie ist die rechte Seite einfacher lesbar oder? Ach ja, weil wir gerade beim Thema „lesbarer Code“ sind – ich rate dringend auch jedem, die Einrückungen nach einer geschweiften Klammer durchzuführen, um die innere Struktur des Codes besser verständlich zu machen. Hierbei merke Dir bitte folgende Regel:

- **Nach jeder öffnenden geschweiften Klammer rücken wir um einen Tabstop nach rechts.**
- **Vor jeder schließenden geschweiften Klammer rücken wir um einen Tabstop nach links.**

Tue Dir diesen gefallen und berücksichtige diese Regel. Sonst bist Du ruck zuck im Codenirwana und findest Dich nicht mehr zurecht!

Auf einen Punkt in Sachen Verzweigung muss ich auf jeden Fall noch eingehen – die Sache mit dem Problem, dass eine Verzweigung „nur“ zwei Pfade – den „Ja“ und den „Nein“ Pfad hat. Was mache ich, wenn ich mehrere unterschiedliche Situationen habe – bspw. wenn ich entsprechend eines Variablenwertes drei verschiedene Zustände unterscheiden möchte? Nun – hier können wir ja die Verzweigungen verschachteln. Sehen wir uns das rechts stehende Struktogramm an. Hier wird die Variable iPosition auf drei verschiedene Zustände geprüft.

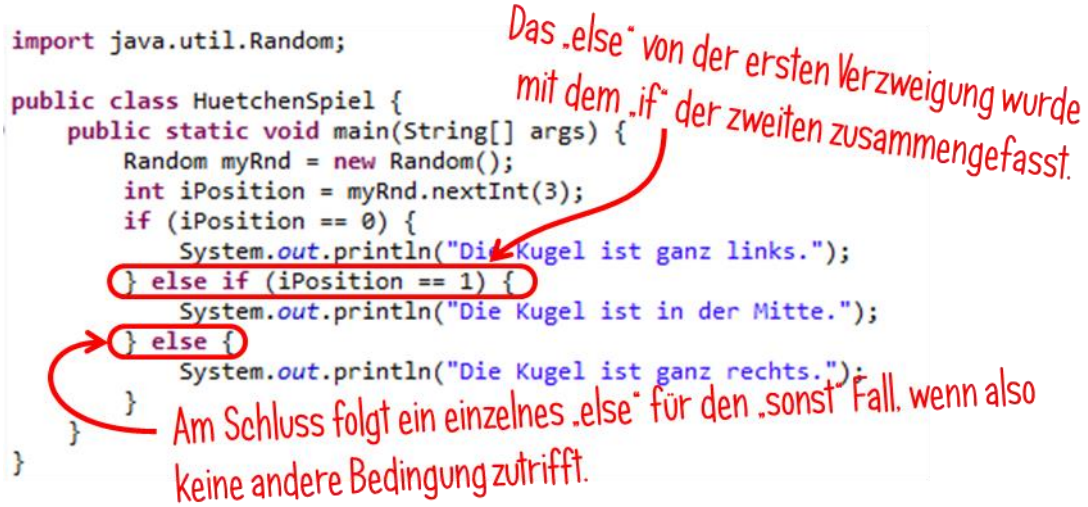




Im Java würde das Programm wie folgt aussehen:

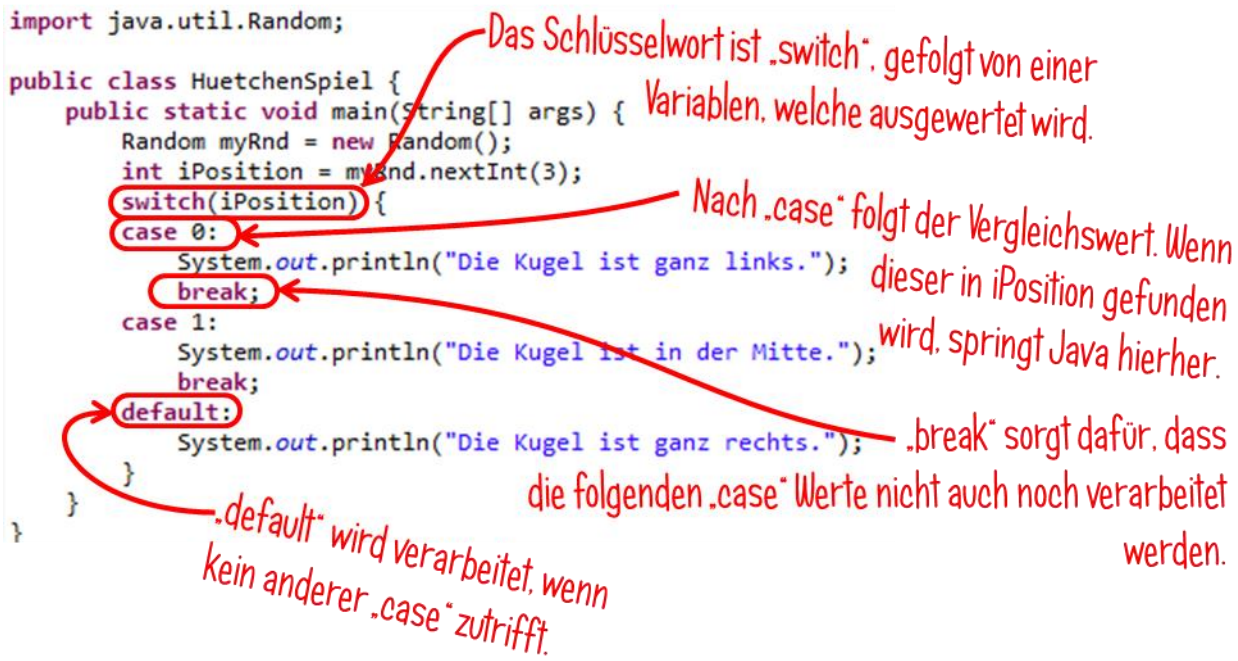
```
import java.util.Random;

public class HuetchenSpiel {
    public static void main(String[] args) {
        Random myRnd = new Random();
        int iPosition = myRnd.nextInt(3);
        if (iPosition == 0)
        {
            System.out.println("Die Kugel ist ganz links.");
        } else {
            if (iPosition == 1) {
                System.out.println("Die Kugel ist in der Mitte.");
            } else {
                System.out.println("Die Kugel ist ganz rechts.");
            }
        }
    }
}
```

Ich überlasse es nun Dir, den Zusammenhang zwischen dem Code und dem Struktogramm herzustellen. Wir werden uns jetzt aber im Folgenden noch zwei weitere Möglichkeiten ansehen, diese Logik in Java umzusetzen. Beginnen wir mit einer Kurzform einer verschachtelten Verzweigung, bei der wir eine aufeinander aufbauende Prüfung umsetzen:



Es wurden in diesem Code also einfach die „else“ Zweige mit den „if“ Zweigen des Folgestatements verknüpft. Dadurch sieht man bei den einzelnen Verzweigungen auch eine Hierarchie – je weiter Oben sie stehen, umso „wichtiger“ sind sie. Wenn bspw. eine obere Bedingung zutrifft, obwohl eine untere Bedingung auch zutreffen würde, geht die untere leer aus. Sie sind somit „exlusiv“.

Java – und die meisten anderen Programmiersprachen – bieten zu diesem Konzept noch eine Alternative, die sogenannte „Mehrfachauswahl“, oder auch „switch/case“ Anweisung. Dieses Konstrukt ist derart verbreitet, das sie auch ein eigenes Struktogrammsymbol bekommen hat. Fangen wir aber zuerst mal mit dem Java Code an:



Gehen wir den Code nochmal schrittweise durch und beginnen mit der folgenden Zeile:

```
switch(iPosition){
```

Hier wird die Variable iPosition geprüft und zwar exakt auf die Werte, die weiter unten im „case“ Block stehen. Diese müssen zwischen der sich öffnenden geschweiften Klammer in dieser Zeile und der passenden Schließenden Klammer stehen. Bis Java Version 6 dürfen als Variablen nur ganzzahlige Datentypen, oder char stehen (was ja eigentlich auch ein ganzzahliger Datentyp ist). Ab Version 7 darf das auch ein String sein. (Achtung! Wenn Du einen Code für andere Rechner schreibst und eine Stringvariable im switch – Statement prüfst, müssen die Rechner Version 7 haben! Sonst stürzt das Programm ab...). Nun folgt der „case“ Block:

```
case 0:
      System.out.println("Die Kugel ist ganz links.");
      break;
case 1:
      System.out.println("Die Kugel ist in der Mitte.");
      break;
```

Hier stehen alle Werte, welche in der Variablen geprüft werden sollen, jeweils nach dem Schlüsselwort „case“ und vor einem Doppelpunkt. Dies heißt aber auch, dass wir hier keine Bereiche prüfen können. Also der Code „`case >0 && < 10:`“ würde in Java nicht funktionieren! Hier musst Du auf eine verschachtelte Verzweigung ausweichen. Es gibt zwar Programmiersprachen in denen die Mehrfachauswahl auf Basis eines Bereiches hinhaut, aber bei den meisten ist es auf einen exakten Wertvergleich ausgelegt.

Nach dem Doppelpunkt kommt nun der Code, welcher verarbeitet werden muss. Dieser läuft nun bis entweder zum Ende des switch Statements, oder wenn vorher ein „break“ gefunden wurde, dann nur bis dorthin. Die „break“ Einträge sind somit absolut wichtig, wenn man nicht möchte dass nach dem „case 0:“ Eintrag nicht auch noch die Ausgabe des „case 1:“ Eintrages ausgeführt werden soll.

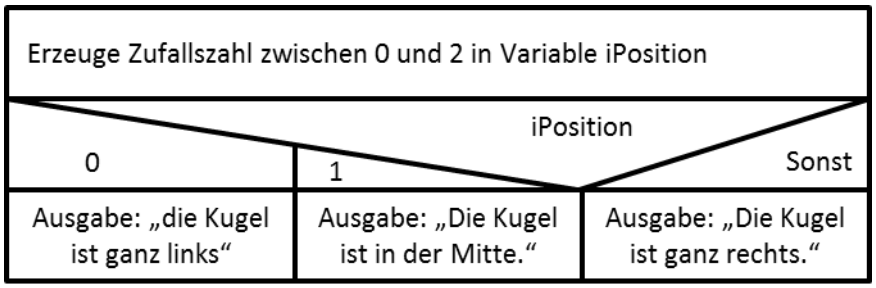
~~Blöde~~ Frage:
Warum haben die Java Macher das „break“ denn überhaupt eingeführt? Der Rechner sollte ja anhand des nächsten „case“ Eintrags wissen, dass hier ein neuer case Bereich anfängt oder?
Antwort: Für den Standardfall mag dies richtig sein. Es gibt aber – zugegebenermaßen seltene – Fälle, in denen man eine hierarchische Abarbeitung haben möchte. Also, wenn wir im obersten case ist, dann muss der gesamte Code durchlaufen werden, im zweiten case Fall alle, bis auf den ersten usw. Um sich diese Möglichkeit nicht zu nehmen, baut das switch Statement darauf, dass es mit „break“ abgebrochen wird.

Am Schluss steht immer der „sonst“ Fall, der „default“ case:

```
default:
      System.out.println("Die Kugel ist ganz rechts.");
```

Wenn alle vorausgegangenen Fälle nicht greifen, dann geht die Anweisung in diesen (optionalen) Defaultzweig rein. Man muss ihn also nicht umsetzen, aber meistens existiert er. Hier brauchen wir auch kein „break“ mehr, da es sich um den letzten case handelt.

Sehen wir uns nun noch das Struktogrammsymbol für die Mehrfachauswahl an. Hier finden wir auch wieder ein Dreieck als Hinweis darauf, dass sich das Programm nun verzweigen wird. Im Dreieck steht nun nicht mehr eine Bedingung, sondern die Variable, welche geprüft werden muss. Auf der linken Seite finden sich die Werte, welche in Java über „case“ abgefragt werden. Der „default“ Fall wird auf der rechten Seite abgehandelt. Die case – spezifischen Anweisungen werden wieder in die nach Unten hin folgenden Spalten gepackt.



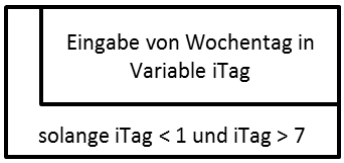
Wir halten also fest:

- Kontrollstrukturen dienen dazu, den Programmfluss zu kontrollieren.
- Um eine programmiersprachenunabhängige Notation zu ermöglichen, wurde das Struktogramm, oder auch Nasi Shneidermann Diagramm eingeführt.
- Die einfachste Kontrollstruktur ist die „Sequenz“, bei der alle Anweisungen in einer Reihe dargestellt und abgearbeitet werden.
- Verzweigungen haben eine Bedingung, welche entweder „true“ oder „false“ sein kann – also ein boolean Datentyp darstellt. Somit hat eine Verzweigung „nur“ zwei Zustände, wobei manchmal auch der „falsch“ – Zweig weggelassen wird, da nur bei einer erfüllten Bedingung eine Aktion durchgeführt werden muss.
- Je nachdem, ob die Bedingung „true“ oder „false“ ist, wird entweder der Wahrzweig oder der Falschzweig abgearbeitet.
- Struktogrammelemente kann man beliebig ineinander verschachteln. Die einfachste Form ist eine verschachtelte Verzweigung, bei der man nun mehr als nur zwei Zustände abbilden kann. In Java existiert für diese Verschachtelung auch eine „Kurzform“, bei der der Falschzweig mit der nachfolgenden Bedingung verknüpft wird.
- Eine Mehrfachauswahl prüft einen Variablenwert auf verschiedene Zustände ab und ist somit eine weitere Möglichkeit, das Programm zu verzweigen.

Ja – das ist ja schon eine ganze Menge! Was uns jetzt eigentlich nur noch fehlt, sind die Schleifen. Schleifen sind (neben rekursiven Methodenaufrufen) die einzige Möglichkeit, Programmbereiche wiederholt aufzurufen.

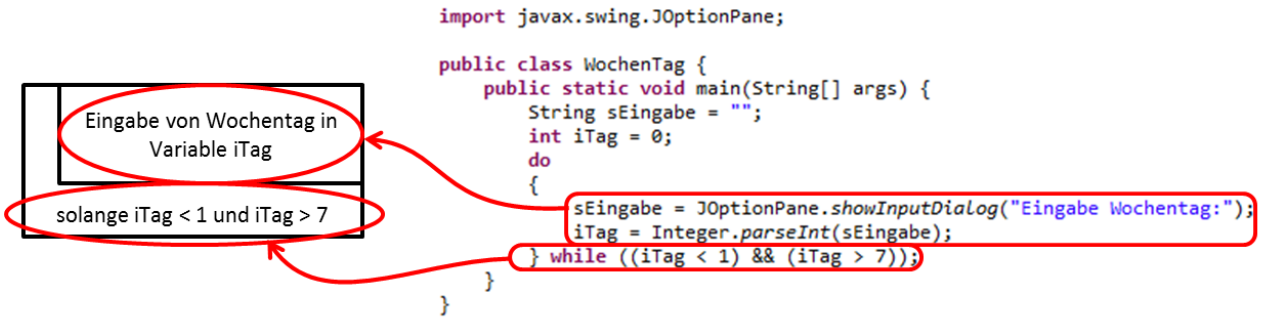
Hierbei unterscheiden wir zwischen drei Grundtypen von Schleifen – die fußgesteuerte, die kopfgesteuerte und die Zählschleife. Doch gehen wir mal Schritt für Schritt vor und beginnen mit der **fußgesteuerten Schleife**. Sie zeichnet sich dadurch aus, dass eine Aktion durchgeführt und danach geprüft wird, ob die Aktion wiederholt werden soll.

Beginnen wir mit dem Struktogrammsymbol – hier wird der Sinn der fußgesteuerten Schleife recht schnell kar. Die Logik soll eine Usereingabe entgegennehmen und auf Validität prüfen. Da die Eingabe (als Ziffer) der Wochentag sein soll, muss diese zwischen 1 und 7 sein. Sollte der User eine andere Zahl eingeben, so wird die Eingabe wiederholt.

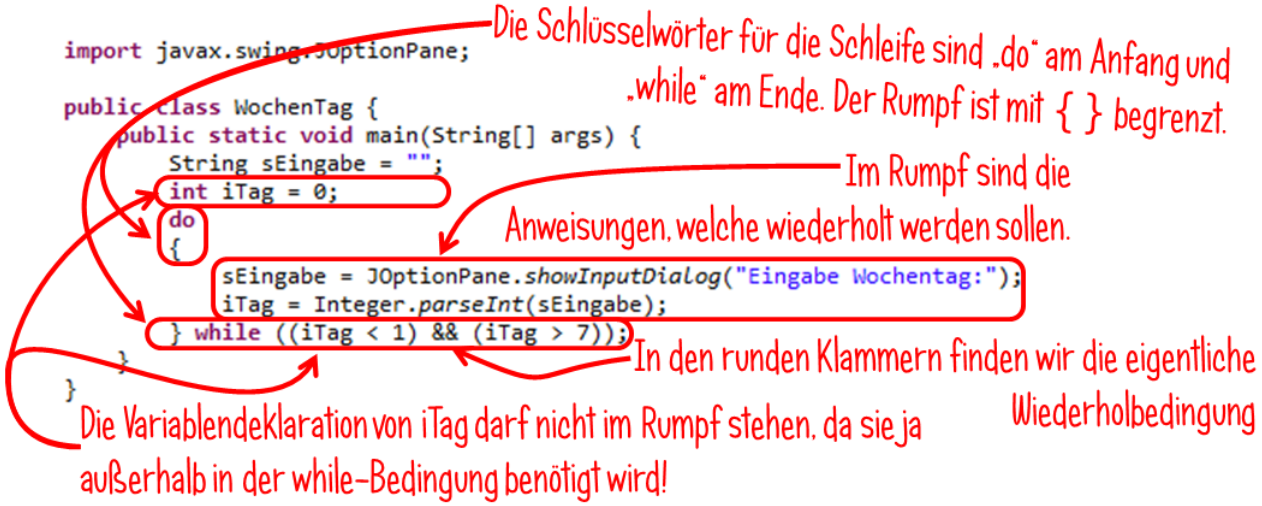


Das Struktogrammsymbol besteht aus zwei ineinander liegenden Rechtecken. Das innere steht für die Aktion – bzw. dies auch wieder beliebig viele verschachtelte Aktionen sein können, das äußere symbolisiert die Wiederholschleife, wobei die Wiederholbedingung sozusagen die Steuerung darstellt und sie hier – wie der Name „fußgesteuert“ schon annehmen lässt – am Ende liegt. Dies ist eine Bedingung wie bei der normalen Verzweigung auch. Es handelt sich um einen boolean Datentyp, der entweder „true“ oder „false“ sein kann.

Soweit, so gut – nun sind wir also bereit für unseren Java Code. Zu Beginn machen wir einfach mal eine Gegenüberstellung des Codes und des Struktogramms:



Wie Du siehst, gibt es eine einfache Entsprechung des Codes und des Struktogramms. Wenn jetzt jemand die Stirn runzelt und sagt – „Warum steht da noch so viel Code über der eigentlichen Schleife?“, dann müssen wir uns den Code wohl oder übel nochmal genauer ansehen. Das folgende Bild sieht zwar ein bisschen überladen aus, aber wir müssen dringend über die einzelnen Punkte sprechen.

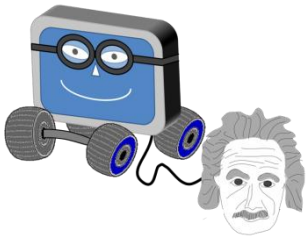


Wie Oben schon gesehen, benötigt die fußgesteuerte Schleife zwei Schlüsselwörter. Eingeleitet wird sie mit „do“. Dadurch weiß der Compiler – „Achtung, jetzt kommt eine fußgesteuerte Schleife“. Danach folgt eine sich öffnende und schließende geschweifte Klammer: der Rumpf der Schleife. Hier stehen all die Befehle drin, welche (gegebenenfalls wiederholt) ausgeführt werden sollen. Abgeschlossen wird die Schleife durch das Schlüsselwort „while“, gefolgt von einer runden Klammer. Diese dient dazu, die Wiederholbedingung zu formulieren. Sie muss entweder „true“ oder „false“ ergeben. Bei „true“ wird die Schleife nochmals durchlaufen und bei „false“ bricht sie ab.
Wichtig ist nun folgende Beobachtung im Code: ich habe die Variable iTag außerhalb der Schleife deklariert! Man möchte jetzt meinen, dass ich sie auch innerhalb der Schleife – also im Schleifenrumpf deklarieren hätte können – das würde aber zu einem Fehler führen. Wir haben bei den Variablendeklarationen ja gelernt, dass die Variablen nur innerhalb der geschweiften Klammern existieren, in denen sie deklariert wurden. Würde iTag nun im Rumpf deklariert werden, so würde die Gültigkeit dieser Variablen vor dem Schlüsselwort „while“ enden, da hier ja die geschweifte Klammer sich schließt. Also muss ich zwangsläufig die Variable iTag vor dem „do“ deklarieren, damit ich sie in der „while“ Bedingung abprüfen darf.

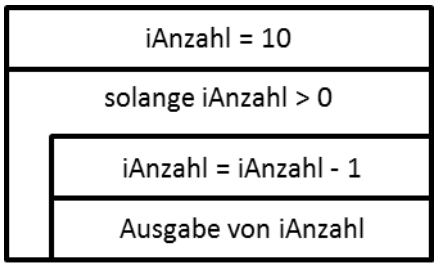
Zusatzinfo:
Noch ein wichtiger Hinweis, zu allen Schleifen: Wenn Du in der Bedingung auf eine Variable prüfst, welche im Rumpf nicht verändert wird, dann hast Du höchstwahrscheinlich eine Endlosschleife programmiert! Also achte bitte darauf, dass die Schleifen auch wirklich terminieren (also „enden“) können! Solltest Du trotzdem mal eine Endlosschleife in Eclipse programmiert haben, dann kannst Du mit dem roten Viereck (meistens unten rechts) das Programm abbrechen. Solltest Du das nicht tun, so kannst Du davon ausgehen, dass der Prozess weiterläuft und nach spätestens dem 3. Programm mit Endlosschleife wird Dein Rechner in die Knie gehen! Im Windows Taskmanager findest Du dann auch eine entsprechende Anzahl von „javaw“ Prozessen.

~~Blöde~~ Frage:
Ich habe in einer anderen Programmiersprache gesehen, dass eine Schleife bei Erfüllung der Bedingung abbricht und nicht wie hier weiterläuft. Gibt es so was in Java auch?
Antwort: Du sprichst hier eventuell von Basic, wo es eine „do until“ Schleife gibt. Solche Schleifen gibt es in Java nicht, wobei das kein großes Problem ist. Wenn wir einfach eine Klammer um die Wiederholbedingung machen und ein Ausrufezeichen davor, dann haben wir den Logikterm negiert. Dadurch wird aus einer Wiederholbedingung einfach eine Abbruchbedingung. Vor diesem Hintergrund ist es aber auch wichtig zu verstehen, dass wir in den Struktogrammen den Leser immer darauf hinweisen, ob wir eine Abbruchbedingung („wiederhole bis xyz eintrifft“) oder eine Wiederholbedingung („wiederhole solange xyz wahr ist“) formuliert haben.

Puh – das ist jetzt mal wieder eine ganze Menge gewesen. Wer Lust hat, sollte sich jetzt mal eine Pause gönnen und die Sache mal sacken lassen. Ich verspreche, wenn ihr die fußgesteuerte Schleife danach nochmal anseht, wird es plötzlich ganz einfach erscheinen.



So, dann sind wir wohl fit für die nächste Schleife! Nach der fußgesteuerten Wiederholschleife folgt nun die **kopfgesteuerte Wiederholschleife**. Mit ein bisschen Fantasie kannst Du Dir ja schon denken, wie das Ding aussehen wird. Kopfgesteuert bedeutet, dass die Kontrolle der Schleife – sprich die Frage, ob wiederholt werden muss – am Anfang gestellt wird und nicht am Ende. Werfen wir einen Blick auf das Struktogramm:



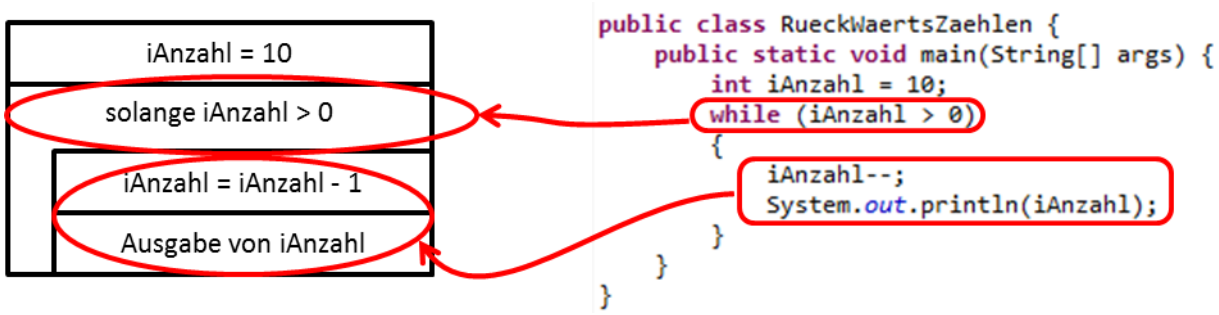
Da ich hier neben der Schleife noch zusätzliche Befehle eingebaut habe ist die erste Aufgabe für Dich: suche die Schleife!

Einfach – oder? Das Layout der kopfgesteuerten Schleife ist die logische Abänderung der fußgesteuerten. Der Bereich in dem die Wiederholbedingung steht, ist einfach nach Oben gerutscht. Das war’s. Es wird also zuerst geprüft und auf dieser Basis festgelegt, ob die Schleife durchlaufen werden soll.

Das führt uns zu folgendem wichtigen Unterscheidungsmerkmal der beiden Schleifentypen:

Eine fußgesteuerte Schleife wird **mindestens einmal** durchlaufen. Bei kopfgesteuerten Schleifen kann es passieren, dass sie nie durchlaufen werden!

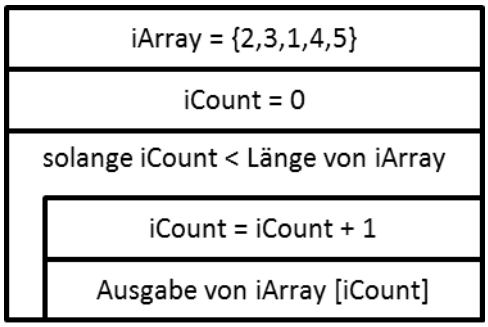
Das liegt einfach daran, dass bei fußgesteuerten Schleifen zuerst durchgeführt wird und danach gefragt wird, ob wiederholt werden muss. Schauen wir nun mal auf den passenden Java Code.



Auch hier finden wir das Schlüsselwort „while“ mit der Wiederholbedingung (bzw. eigentlich „Durchlaufbedingung), allerdings nicht am Ende, sondern eben am Anfang des Statements. Das „do“ brauchen wir hier nicht, da das „while“ ein ausreichender Hinweis für den Compiler darstellt. Weiterhin noch der Hinweis, dass nach der schließenden Runden Klammer nach „while“ kein Semikolon („;“) folgt! Es ist ein häufiger Anfängerfehler, dass hier ein Semikolon eingetragen wird und somit das Statement in der ersten Zeile schon abgeschlossen wird. Der Rumpf befindet sich dann zwischen der schließenden runden Klammer und dem Semikolon – er ist also dann leer. Solche Fehler findet man als Anfänger nur sehr schwer – insofern bitte hier aufpassen! Ansonsten gibt es zur kopfgesteuerten Schleife nicht mehr zu sagen, da sie na im Prinzip nur eine Umkehrung der fußgesteuerten ist.

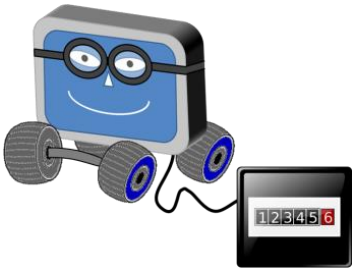
Nun, damit wären wir eigentlich fertig – wir bräuchten streng genommen keine weiteren Schleifen. Es gibt aber noch eine dritte Schleife, welche vom Prinzip her aber nichts anderes ist, als eine leicht erweiterte kopfgesteuerte Wiederholschleife.

Sehen wir uns mal folgendes Struktogramm an:

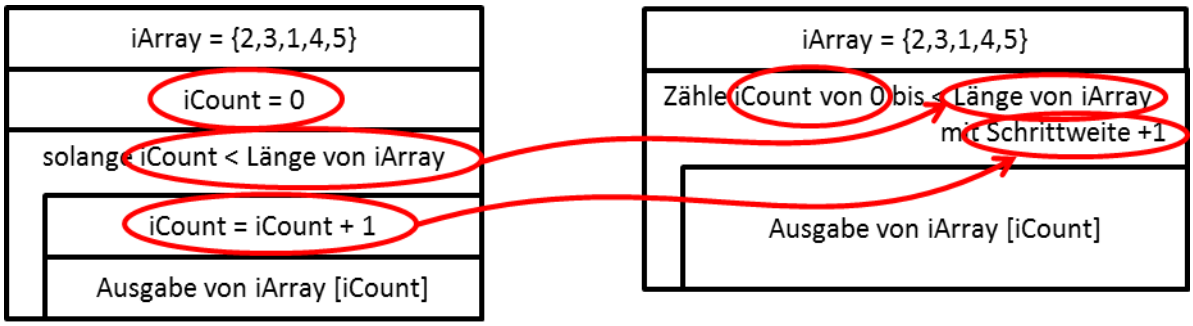


Das ist erst mal nichts anderes, als unsere Schleife von Oben, bei der wir einfach aufwärts zählen und den Inhalt eines Arrays ausgeben. Man muss jetzt keine seherischen Fähigkeiten haben um sich zu überlegen, dass solche Konstrukte relativ häufig vorkommen können.

Um diesem Umstand entgegen zu kommen, haben die Java Macher die **Zählschleife** integriert. Sie ist tatsächlich nichts anderes, als eine verkürzte Form der links stehenden Schleife – so dass wir sie als eine Einheit betrachten können.



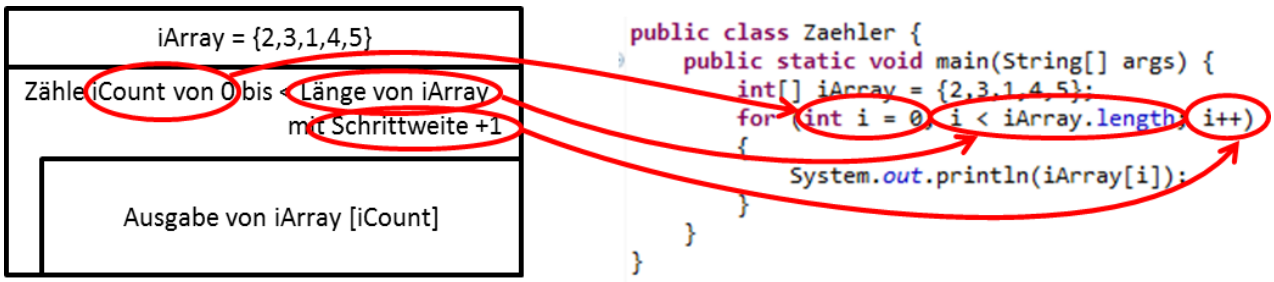
Schauen wir uns mal das Symbol für diese Schleife an und zwar gleich im Zusammenhang mit dem links dargestellten Beispiel auf Basis einer kopfgesteuerten Wiederholungsschleife:



Wie Du siehst, hat sich eigentlich nicht viel verändert – es werden lediglich die einzelnen Elemente, die wir für das Zählen brauchen in den Kopf der Zählschleife eingesetzt. Auf dieser Basis können wir auch die wichtigsten Eigenschaften von Zählschleifen benennen:

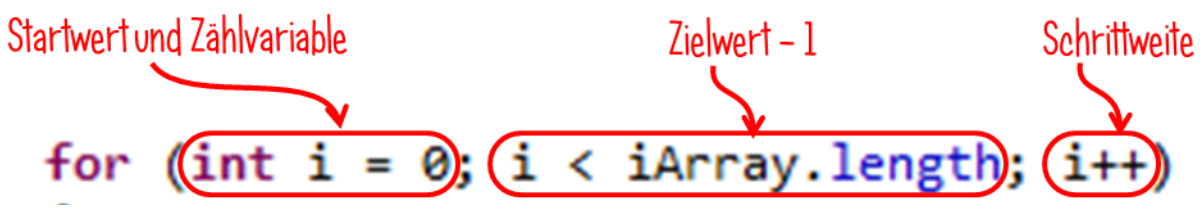
- Zählschleifen brauchen eine Zählvariable, in welcher der aktuelle Zählerstand abgelegt wird.
- Zählschleifen sind kopfgesteuert – das heißt, dass wir Situationen erleben können, in denen der Rumpf nie durchlaufen wird (wenn bspw. im obenstehenden Beispiel das Array die Länge 0 hätte)
- Zählschleifen haben eine Zählbedingung, welche im „true“ Fall zum Durchlaufen des Rumpfes führt und im „false“ Fall zum Abbruch.
- Zählschleifen benötigen im Regelfall eine Angabe, wie hoch die Schrittweite sein muss – welche auch negativ sein kann. Solltest Du ein Struktogramm mit einer Zähschleife finden, bei denen keine Schrittweite angegeben ist, dann kannst Du von +1 ausgehen.

Jetzt fehlt nur noch der Java Code:



Es ist nun nicht wirklich verwunderlich, dass wir auch im Java Code die Elemente der Zählschleife wieder im Kopf der Schleife finden.

Um das Ganze nochmal etwas deutlicher darzustellen, hier nochmal der Kopf der Zählschleife im Zoom:



Die drei Bereiche „Startwert“, „Zielwert – 1“ und „Schrittweite“ sind also mit Semikolons getrennt.

Folgendes gilt es hierbei zu beachten:

Startwert: Hier wird in der Regel eine eigene Zählvariable deklariert und mit dem Startwert initialisiert. Es hat sich eingebürgert, dass diese Variable „i“ wie „Integer“ heißt. Die Variable ist übrigens nur im Rumpf der Zählschleife gültig (obwohl sie ja streng genommen außerhalb der geschweiften Klammer deklariert wurde).

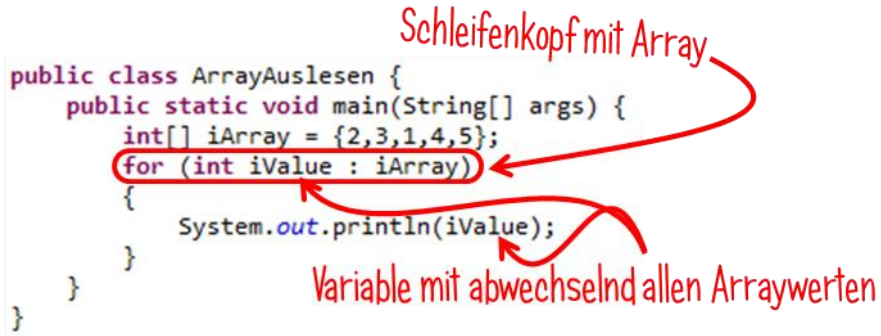
Zielwert: Da wir beim Programmieren immer bei 0 anfangen zu zählen, muss eine Schleife die 10-mal durchläuft bis **kleiner** 10 laufen – also von 0 bis 9. Dies passt auch hervorragend zur Tatsache, dass ein Array der Länge 10 die Indexpositionen 0 bis 9 aufweist!

Schrittweite: Hier kann eine beliebige Aktion stehen. Im einfachsten Fall ist es i++ für das Hochzählen um 1. Es kann aber auch in bspw. zweierschritten gezählt werden. Dann würde an dieser Stelle `i+= 2` stehen. Wichtig ist, dass die Erhöhung – obwohl sie im Kopf der Schleife steht – am Ende des Rumpfes durchgeführt wird.

Wir halten also fest:

- Schleifen dienen dazu, Programmbereiche wiederholt auszuführen. Hierbei müssen wir aufpassen, dass wir keine Endlosschleifen produzieren.
- Wir unterscheiden zwischen kopfgesteuerten und fußgesteuerten Wiederholungsschleifen.
- Fußgesteuerte Schleifen werden mindestens einmal durchlaufen.
- Kopfgesteuerte Schleifen können unter Umständen nie durchlaufen werden – je nachdem wie die Bedingungen formuliert sind.
- Zählschleifen sind Sonderformen von kopfgesteuerten Schleifen und benötigen immer eine Zählvariable, eine Laufbedingung und eine Schrittweitendefinition.

Zum Schluss will ich noch einen Spezialfall von Schleifen im Zusammenhang mit dem Auslesen eines Arrays eingehen. Sieh Dir einfach mal folgenden Code an und versuche den Sinn dahinter zu verstehen:



Diese verkürzte Form dient dazu, ein komplettes Array auszulesen. Hierbei wird in die Variable „iValue“ iterativ, beginnend bei der Indexposition 0 jeder Wert eingetragen und danach die Schleife durchlaufen. Intern behandelt Java diesen Code wie eine normale Schleife auch – da das Auslesen von Arrays jedoch so häufig vorkommt, wurde hier eben diese Kurzform geschaffen, um dem Java Programmierer entgegenzukommen.

Soo, das war es vorerst mal – ich hoffe, dass ich mit diesem Text das Thema Java ein wenig aufhellen konnte. Wenn nicht, dann schlage ich vor – nochmal lesen und vor allem sich auch mal an den Rechner trauen um die ganzen Sachen mal auszuprobieren. Nur wer das Ganze intensiv übt, wird ein Meister ☺