Disclaimer:	Dieses Dokument wurde zur Unterstützung des Informatik 2 Tutoriums (Schwerpunkt: Java) von Luca Dreiling im Sommersemester 2019 erstellt. Es handelt sich hierbei um ein inoffizielles Dokument, für die Korrektheit wird nicht garantiert und ist Unabhängig zur jeglichen Vorlesungen. Bei Unklarheiten oder Fehlern ist Feedback erwünscht.
Für vertieftes Wissen:	Buch (als HTML-Version) "Java ist auch eine Insel"vom Rheinwerk Verlag, (bedingt) das Buch "Clean Code"von mitp-Verlag, stackoverflow.com und eigenes Codieren. (V 1.1)

Werte(Literale)

Literal	Literale Zeichen/Ausdrücke, die im Programm einen konkreten Wert repräsentieren.
Zahlen	Werden durch logischerweise durch Zahlen repräsentiert. Bei Nachkommastellen für Doubles/Float wird der Punkt verwendet! Man schreibt 0.5 und nicht 0,5! Weist man einem Double/Float eine ganze Zahl durch können Nachkommastellen weggelassen werden.
Boolsche Werte	Werden durch `true` und `false` repräsentiert.
Buchstaben/ Character	Ein (Unicode-)Zeichen. Repräsentiert durch Zeichen oder Unicode-Nummer in Apostrophen `'A'` bzw. `'\u0041'` oder als Dezimalwert `65`.
Strings	Eine Zeichenkette. Eine beliebige Kette von Zeichen zwischen zwei Gänsefüßchen. Die Kette kann leer sein. Zum Beispiel: `""`, `"Java"`oder `"Java Eclipse 123"`.
`null`	„Nichts“. Der initiale Wert eines Objekts, wenn einer Variable noch kein Wert zugewiesen wurde. Dieser Fall muss oft abgefangen werden, da man mit „Nichts“nichts machen kann, was für gewöhnlich zum Programmabsturz führt.
Objekt	Ein Objekt ist ein nicht-primitives Konstrukt aus verschiedenen Datentypen und Funktionen um mit ihnen umzugehen. Alle Eigenschaften für ein Objekt werden in einer Klasse gesammelt. Anhand dieser Klasse können dann unendlich viele unabhängige Objekte, ähnlich wie bei den primitiven Datentypen, erstellt werden. Ein Beispiel dafür ist die Klasse String, die String-Objekte erstellt.

Primitive Datentypen	
Definition	Primitive Datentypen sind die in Java eingebauten Datentypen, die Standardwerte speichern können. Das sind Boolsche Werte, Ganzzahlige und Rationale Zahlen, Unicode-Zeichen (KEINE Strings) und null. Sie werden in Eclipse, im Gegensatz zu nicht primitiven Werten, fett als sog. Keyword hervorgehoben. (Aufpassen: Es sind nicht die einzigen Keywords).
Ganze Zahlenwerte	
Integer (**`int`**)	Standardtyp für ganzzahlige Werte. Es gibt, unter Berücksichtigung des vorhandenen Speichers und des Wertebereiches, noch andere Typen mit kleineren/größeren Wertebereichen und somit kleinerem/größerem Speicherbedarf. Standardmäßig nutzt man den normalen Integer.
Wertebereiche (von klein nach groß)	Byte (**`byte`**); Short Integer (**`short`**); Integer (**`int`**); Long Integer (**`long`**)
Rechenoperationen(`+`, `-`, `*`, `/`, `%`)	
Addition und Subtraktion (`+`, `-`)	Funktionieren ganz intuitiv. (Bei extrem großen positiven/negativen Ergebnissen muss der Wertebereich beachtet werden.)
Inkrementation/ Dekrementation (**`x`**`++`, **`x`**`--` bzw. `++`**`x`**, `--`**`x`**)	Addiert bzw. Subtrahiert `1` zu einer Variable hinzu. Meist nur als einzelnes Statement in einer Zeile. Stehen die Zeichen vor einer Variable (`++`**`x`**, `--`**`x`**) wird die Variable VOR der Weiterverarbeitung um eins erhöht (entspricht `(`**`x`**`+1) ...`) bzw. (`(`**`x`**`-1) ...`). Stehen sie nach der Variable, dann die `1` wird NACH der Verarbeitung verrechnet (entspricht `(...)+1`) bzw. (`(...)-1`).
Multiplikation (`*`)	Auch intuitiv, darf aber nicht wie in der Mathematik weggelassen werden: `5x` muss als `5 * x` geschrieben werden.
Division (`/`)	Hier muss man einiges beachten: Man darf nicht durch `0` teilen! Es muss sicher gestellt werden, dass das nicht passieren kann (z.B. durch eine `if`-Bedingung). Außerdem wird im Wertebereich dividiert. Bei Integern - also im Ganzzahligen Bereich - das Ergebnis wird immer zur `0` hin gerundet (z.B. `5/2 = 2` und `(-1)/2 = 0`).
Remainder/ "Modulo"(`%`)	Entspricht dem normalen Rest rechnen bei positiven Werten. Im Negativen wird der Betrag Modulo genommen und anschließend negiert. Z.B.: -5 % 3 = -2 (funktioniert alles auch bei Rationalen Werten).

Rationale Zahlenwerte	
Double (**double**)	Kann im Gegensatz zu Integerwerten auch Gleitkommazahlen speichern. Meistens wird direkt Double verwendet, da es genauer als Float rechnet.
Float (**float**)	Kleinerer (und etwas ungenauerer) Wertebereich als der eines Doubles.
Boolsche Algebra	
Boolscher Wert (**boolean**)	**true** oder **false** . Sie können in Variablen gespeichert, aus Boolschen Ausdrücken errechnet oder direkt verwendet werden.
Boolsche Operatoren	`&&` ≙ UND, `||` ≙ ODER, `!` ≙ NICHT; `<` ≙ KLEINER; `>` ≙ GRÖßER, `>=` ≙ GRÖßERGLEICH, `<=` ≙ KLEINERGLEICH, `==` GLEICH (das Selbe), `!=` ≙ UNGLEICH.
Boolsche Ausdrücke	Werden im Programm vor der Verwendung zu EINEM konkreten Boolschen Wert ausgewertet. (z.B. mit der Verwendung einer boolschen Variable `b = false`: `(!b && true ) || b = (!false && true ) || false = (true && true ) || false = true || false = true`. Der Ausdruck hat also den Wert **true**).
Operator '==' und `.equals()`	Der Operator schaut, ob die beiden Werte die Selben sind, also die Referenzen gleich sind. Bei primitiven Typen sind gleiche Werte immer auch die Selben, der Code `int x= 42; System.out.println(x== 42);` liefert **true** zurück. Bei Objekten (z.B. Strings) können zwei Objektinstanzen den gleichen Wert haben, sie sind aber nicht die Selben. `Thread t1 = new Thread(); Thread t2 = new Thread(); System.out.println(t1 == t2);` ergibt die Ausgabe **false**, da beide Objekte zwar die gleichen (initialen) Werte besitzen, es sich hier aber um zwei verschiedene Objekte mit eigenen Referenzen handelt. Ist eine entsprechende `.equals()`-Methode für ein Objekt implementiert, erzeugt der Code `Thread t1 = new Thread(); Thread t2 = new Thread(); System.out.println(t1.equals(t2));` die Ausgabe **true**. Für Objekte sollte generell immer die `.equals()`-Methode verwendet werden. Strings sind ein Spezialfall, bei dem '==' manchmal(!) funktioniert, aber nicht zuverlässig ist (Stichwort: String-Interning).

Buchstaben	
Character (char)	Der Character (bzw. auch das Byte) ist ein besonderer Wert. Es ist eigentlich ein Unicodezeichen, der als Zahl gespeichert wird. Dieser Wert kann dadurch auch als positiver Integer interpretiert werden. Nutzt man einen Rechenoperator wird wie bei Integern gerechnet und das Ergebnis normalerweise als Zahl und nicht als Unicode zeichen interpretiert. (z.B. `char a = 65` ≙ `a = 'A'` und `a + 1 = 66`).
Byte (byte)	Funktioniert wie beim Charcter, aber mit einem kleineren Wertebereich von `-128` bis `127`.
KEIN! Primitiver Datentyp, trotzdem extrem wichtig (Stichwort: Objekte)	
Strings (`String`)	Speichert eine Zeichenkette von Charactern. Damit können Wörter und Texte im Programm dargestellt werden.
Addition/Konkatenation	Strings unterstützen die Addition, wobei alle primitiven Datentypen und fast alle Objekte mehr oder weniger sinnvoll zu einem String umgewandelt und aneinander gehangen werden. Das nennt man Verkettung oder Konkatenation.

Typkonvertierung(Casten)	
Typkonvertierung	Allgemein ist das die Änderung eines Datentyps. Dabei gibt es zwei Arten: Die impliziete und explizite Typkonvertierung. Die Änderung eines Datentyps geschieht dann, wenn mit verschiedenen Datentypen gerechnet wird oder ein Wert in einem anderen Datentyp gespeichert wird. Wichtig ist, dass diese Umwandlung erst bei der Kollision von Typen geschieht. (z.B.: Dividiert man zwei Integer und weist das Ergebnis einer Double-Variable zu, wird erst im Integer-Wertebereich dividiert und dieses Ergebnis zu einem Double konvertiert. Also `x= 2/5` ⇒ `x = 2.0`)
implizite Typkonvertierung	Hier geschieht die Konvertierung automatisch, da die Konvertierung in ein genaueres Format erfolgt, somit auch keinerlei Informationen verloren gehen. Zum Beispiel kann man problemlos einer Double-Variable Integerwerte zuweisen. Die Faustregel ist: Implizit ist es dann, wenn man genauer wird und das Ergebnis hat immer den genausten Datentyp.
implizit mögliche Konvertierungen	Byte → Short → Integer → Long → Float → Double und Character → Integer → Long → Float → Double. Booleans können nicht konvertiert werden. (Natürlich können Bytes und Character beispielsweise direkt zu Floats umgewandelt werden)

explizite Typkonvertierung	Möchte man von einem womöglich genaueren Wert auf einen ungenaueren Wert (z.B. Double zu Integer) wechseln gehen womöglich Informationen verloren. Die Umwandlung (der Cast) geschieht explizit, damit man sich bewusst wird, dass hier Informationen verloren gehen. Man schreibt den gewünschten Datentyp in Klammern vor den Wert, der konvertiert werden soll. Aber Achtung!: Man muss sich immer bewusst sein was man konvertiert, da der Cast wie ein Vorzeichen funktioniert: `(int) 1.5+0.5 = 1+0.5 = 1.5` → Das Ergebnis ist kein Integer. (int) 1.5 + (int) 0.5 = 1 + 0 = 1 → Das Ergebnis ist ein Integer, ist aber durch die ungünstige Rundung falsch. (int) (1.5 + 0.5) = (int) (2.0) = 2 → Richtiges Ergebnis. Auch wichtig ist, dass immer zur 0 hin gerundet wird, die Nachkommastellen einfach weggelassen werden. Also ist (int) 1.999... = 1 und (int) -0.001 = 0.
explizit mögliche Konvertierungen	Byte ← Short ← Integer ← Long ← Float ← Double und Character ← Integer ← Long ← Float ← Double. Zusätzlich noch Byte ↔ Char ↔ Short. Booleans können wieder nicht konvertiert werden.

Begriffe zum Projektaufbau und Benennung

Projekt	Im Prinzip das gesamte Programm.
Paket	Ordnerstruktur, um Klassen zu gruppieren. Die Benennung ist etwas umfangreich. Wird nichts vorgegeben können sie in Kleinbuchstaben sinnvoll benannt werden.
Klasse	Beinhaltet Eigenschaften und Methoden zu einem Objekt (Stichwort: Objektorientierte Programmierung). Die Anfangsbuchstaben der Wörter werden groß geschrieben, z.B. Main.java oder MeinProgramm.java
Main-Methode	'`public static main(String[] args) {...}`'. Der Einstiegspunkt des Programms. Jede Java-Application benötigt eine Main-Methode.

Begriffserklärungen & Semantik & Programmstruktur

Ausdruck/Statement/ Befehl	Eine Wertzuweisung einer Variable oder ein Funktionsaufruf. Darauf folgt ein Semikolon, um das Ende des Ausdrucks zu kennzeichnen.
Semikolon (`;`)	Endet ein Statement.

Compiler-Fehler	Fehler der vor Programmstart vom Compiler erkannt wird. Dazu zählen Typkollisionen (z.B. `boolean b = 3;`), nicht geschlossene Blöcke,... Eclipse unterkringelt solche Fehler rot. Protipp: Wenn man mit der Maus auf die markierte Stelle zeigt, wird eine kleine Information angezeigt, die den Fehler genauer beschreibt und Lösungen anbietet. Für das Beispiel der Typkollision wird 'Type mismatch: cannot convert from int to boolean' angezeigt.
Runtime-Error/ Exception	Fehler die während der Durchführung des Programms auftreten und vom Compiler nicht vorher absehbar sind. Dazu zählen Zugriffe auf nicht existente Objekte oder Stellen im Array, Fehler beim Lesen/Durchsuchen von Dateien,... Diese Fehler werfen eine Art Fehlermeldung, die Exception. Sie enthält Informationen über den Fehler und ermöglicht es diese Fälle zu handhaben.
Variablen**deklarartion** (<Datentyp> <Variablenname>)	Erstellt eine neue Variable, mit dem Namen <Variablenname> weist ihr aber noch keinen Wert, sondern nur ihren <Datentyp> zu. Man sag dem Compiler „Es gibt eine Variable <Variablenname> vom Typ <Datentyp>“.
Initalisierung/ Wertzuweisung (<Variablenname> = <Wert>)	Weißt einer Variable von einem primitiven Typ einen konkreten Wert (eines Literals) zu. Ist das die erste Wertzuweisung nennt man das Initialisierung Initialisiert man eine Variable nicht, nutzt aber sie im Programm kommt es zu einem Fehler/Programmabsturz.
Instanzierung	Entspricht der Initalisierung bei Objekten.
Deklaration und Initalisierung (<Datentyp> <Variablenname> = <Wert>)	Man fasst meist Deklaration und Initialisierung in einem Statement zusammen. So vermeidet man Fehler durch nichtinitialisierte Variablen.
Funktionen/Methoden (<Modifier> <Rückgabetyp> <Funktionsname> (<Parameter>,...) {...}	Eine Funktionen ist eine Sammlung von, in einem Block zusammengefassten, Befehlen. Dadurch ersetzt man viele Befehle durch einen Befehl (Funktionsaufruf) und kann so diese Codezeilen an mehreren Stellen einfach aufrufen. Das führt letztlich zu einer übersichtlicheren Programmstruktur. Funktionen geben meistens einen Wert zurück, der dann zum Beispiel einer Variable zugewiesen werden kann. Ist das der Fall, muss an jeder möglichen letzte Zeile der Funktion ein Wert vom Typ <Rückgabetyp> mittels „**return** <Rückgabetyp>;“zurückgegeben werden. Ein Sonderfall ist der <Rückgabetyp> **void**. Hier wird kein Wert zurückgegeben. Hier braucht man keine return-Anweisung, kann aber mit `return;` sofort aus der gesamten Funktion heraus springen. (Modifier werden an anderer Stelle behandelt. Funktion und Methode sind streng genommen keine Synonyme, werden aber meist so verwendet.)

Parameter & Argumente	Parameter sind Variablen, die Im Header einer Funktion deklariert werden. Sie können den Namen einer Klassenvariable haben und Überschreiben diese dann (Siehe „`this.<Variable>`“). Argumente sind die Werte, die einer Funktion beim Aufrufen übergeben werden.
Konstruktor `(<Access-Modifier> <Klassenname>(...) {...}`	Bei der Verwendung von Objekten werden Konstruktoren verwendet, um neue Instanzen von diesen Objekten zu erstellen. Dabei werden entsprechend alle Werte initialisiert und Speicherplatz freigeräumt. Man erkennt eine Konstruktorfunktion an ihrem Namen, die dem Klassennamen entspricht und dem fehlenden Rückgabetyp. Auch wenn kein Konstruktor vorhanden ist, gibt es immer einen parameterlosen Konstruktor, den Standardkonstruktor. Konstruktoren können für jedes Objekt nur ein Mal aufgerufen werden.
Blöcke ({ und }) (angedeutet durch {...})	Dient zur Einteilung des Programms in Zusammenhanggebiete. Zeilen, die beispielsweise in einem `if (...) {...}`-Block stehen, werden alle hintereinander ausgeführt, sofern die `if`-Bedingung erfüllt ist. Da gilt genauso für Anweisungen, die in einem Funktionsblock stehen. Bei korrekter Formatierung, sollte der Inhalt eines Blockes eingerückt sein (Vgl. Codeformatierung mit dem Shortcut Strg + Shift + F).
Klassenvariablen/ Globale Variablen	Variablen die auf einer Ebene mit Funktionen stehen. Auf sie können in allen Funktionen der Klasse zugegriffen werden.
Scope/Existenz	Variablen „existieren“ nur in einem Block (und den darin beinhalteten Blöcken). Das bedeutet: In zwei Funktionen kann eine Variable int result erstellt werden, die sich nicht beeinflussen. Auf eine Globale Variable kann in jeder Funktion zugegriffen werden. Man hat jedoch die Möglichkeit diese Variablen zu überschreiben. Hat man eine Klassenvariable `int x` kann man in einer Funktion eine Variable `double x` definiert werden. In diesem Fall wird nur für diese Funktion die Variable vom Typ Double genutzt.
Pointer	Existieren in Java strng genommen NICHT, dort gibt es nur Referenzen. In anderen Programmiersprachen (z.B. C++) wird eine Variable an einer Speicheradresse gespeichert, um auf diesen Speicherplatz zuzugreifen, wird zu jeder Variable ein Zeiger(Pointer) auf seine Speicheradresse gespeichert. Dadurch können zwei Variablen den gleichen Wert haben, aber nicht die Selben sein. (Pointer meistens werden trotzdem verwendet.)
Referenz	Die in Java verwendete Alternative zu Pointern, sie zeigen viel mehr auf ein Objekt statt auf einen Speicherplatz. Der Umgang mit den Variablen und was genau übergeben wird von Java relativ gut übernommen. Grob: Primitive Datentypen werden generell per Wert übergeben und Objekten mit Referenzen.

Dos and Don'ts	
Kommentieren(!)	Das wichtigste neben dem Code ist eine ausführliche Kommentierung (in den Übungen gibt es ohne Kommentare Abzug). Ein Code den niemand versteht nützt nichts. Oft wird der Code von mehreren Leuten zusammen geschrieben oder mit veröffentlicht. Vor der Abgabe solltet ihr für die meisten Funktionen (abgesehen von Gettern und Settern) und in längeren Methoden das grobe Vorgehen erklären. Die Regel ist: Sollte einem außenstehenden der Code vorliegen, sollte er ihn durch die Kommentare auch verstehen.
Beständigkeit	Nutzt eine einheitliche Formatierung, um den Code übersichtlicher zu gestalten (und haltet euch daran). Zum Beispiel (aber keine Pflicht): Leerzeichen vor und nach einem Operator; '{' am Ende einer Zeile und '}' in eine eigene;...
Sinnvolle Benennung	Variablen und Funktionen nützen nichts, wenn man nicht weiß wofür sie genutzt werden. Deswegen sollten IMMER sinnvolle Bezeichnungen genutzt werden (im Besten Fall auf Englisch). Bei Variablen wird das erste Wort kleingeschrieben, jedes bei jedem weiteren wird der erste Buchstabe groß geschrieben (also z.B.: `myVariable`). Bei der Bennenung muss man beachten, dass sie keinem Keyword entsprechen und nicht mit Zahlen oder Sonderzeichen enthalten. Zählvariablen von kurzer Existenz (also z.B. in Schleifen) werden i.d.R. nicht ausführlich benannt, meistens i oder j. Konstanten werden komplett groß geschrieben und die Wörter mit einem '_' getrennt (z.B. `A_CONSTANT`). Funktionen werden wie Variablen benannt, jedoch sollte das erste Wort ein Verb sein (z.B. `update()` oder `moveCarToDestination()`).
Kurze Codezeilen	Java-Codezeilen werden mit einem Semikolon beendet, sie nicht sensibel für Zeilenumbrüche. Falls eine Zeile zu lang wird sollte sie an geeigneter Stelle mit einem Zeilenumbruch gekürzt werden. In der Literatur ist die Zeilenlänge mit etwa 80-120 Zeichen angegeben. Auch Leerzeilen sind nichts schlechtes.
Kompakter Code	Nach Möglichkeit sollten Boolsche Funktionen recht kurz oder verständlich gehalten werden. Hat man eine Funktion, die z.B. anhand einer Boolschen Bedingung, wahr oder falsch zurückgibt, kann man sie folglich schreiben: '`if(<Bedingung>) { return true;} else{ return false;}`' aber kürzer und besser ist '`return<Bedingung>;`'.

Sinnvolle Codestruktur	Tendenziell kann man sich an folgende Struktur halten: Zuerst Konstanten, Variablen, der Konstruktor, die Methoden und dann die Getter und Setter. Die Methoden/Variablen an sich sollten sinnvoll gegliedert sein. Zwei Methoden, wobei eine die andere aufruft sollten logischerweise möglichst nahe beieinander stehen. Wenn mehrere Methoden ähnlich aufgebaut sind, zum Beispiel beide ein Interface verwenden, sollte die Reihenfolge der Interface-Methoden gleich sein.
`<Boolscher Ausdruck> == true`	Generell sollte ein `<Boolscher Ausdruck>` nie mit **`true`**/**`false`**verglichen werden, da der boolsche Ausdruck bereits entweder als wahr oder falsch ausgewertet wird. Also wird **`if`**`(<Boolscher Ausdruck> == true)` zu **`if`**`(<Boolscher Ausdruck>)` und **`if`**`(<Boolscher Ausdruck> == false)` zu **`if`**`(!<Boolscher Ausdruck>)`.

Keywords, Konstruktionen und deren Nutzen

Allgemein	Keywords sind Worte, denen eine besondere Funktion zugewiesen wird. Sie können also nicht als Variablennamen, etc. verwendet werden.
Primitive-Typnamen	Ansonsen kann man Variablen von Typen nicht unterscheiden.
`if` `(<Bedinung>) {...}`, **`else if`** `(<Bedingung>) {...}`, **`else`** `{...}`	Ein **`if-else`**-Entscheidung besteht aus einem **`if`**-Block und optional weitere **`else-if`**-Bedinungen mit einem abschließenden **`else`**-Block. Führt den if-Block aus, wenn die Bedingung erfüllt ist, das gilt genauso für weiter **`else-if`**-Bedinungen, sofern vorherige Bedingungen nicht erfüllt wurden. Der **`else`**-Block wird ausgeführt, falls keine der vorherigen Bedingungen zutraf. Wurde eine Bedinung erfüllt und der dazugehörige Block ausgeführt, werden alle weiteren Bedingungen und Blöcke der **`else-if`**-Entscheidung ignoriert.
`for` `(<Zählvariable>; <(boolsche) Abbruchbedinung>; <Veränderung>) {...}`	Der zugewiesene Block wird solange wiederholt, bis die `<Abbruchbedinung>` nicht mehr erfüllt ist. Zusätzlich kann man eine Variable als `<Zählvariable>` definieren, die nur im folgenden Block existiert. Am Ende der Ausführung wird `<Veränderung>` durchgeführt. Meistens wird hier die Zählvariable verändert. Tipp: Bleibt bei der Abbruchbedingung konsistent! Nutzt entweder nur < bzw. > oder nur <= oder >=, da das vermutlich einer der häufigsten Leichtsinnsfehler beim Programmieren ist.

	(Klausurwissen: <Zählvariable>, <(boolsche) Abbruchbedinung> und <Veränderung> können weggelassen werden. „**for** (; ;) {...}“ist gültiger Code (Endlosschleife). Außerdem kann man mehrere Zählvariablen des gleichen Typs initalisieren, indem man einmal den Typ angibt und alle Variablen durch ein Komma trennt (also: „**for**(**int** **i** = 0, **j** = 0, **h** = 2; ...)“). Mehrere <Veränderungen> können auch mit Kommas getrennt werden (z.B. “**for**(...; ...; **i**++, **h**--)“).
while(<(boolsche) Abbruchbedinung>) {...}	Wiederholt den Block solange, bis die Abbruchbedingung nicht mehr erfüllt ist. Ähnelt der **for**-Schleife sehr und lässt sich in diese überführen, für den Abbruch muss selbst gesorgt werden.
do{...} **while**(<(boolsche) Abbruchbedinung>);	Eine while-Schleife, wobei der Block auf jeden Fall mindestens ein Mal ausgeführt wird. Zu beachten ist, dass eine nicht erfüllte Abbruchbedingung eventuell zu Fehlern (z.B. **null**-Pointer) führen kann, die man in der **while**-Schleife abfangen muss. Achtung: Das Semicolon am Ende nicht vergessen!
break;	Der **break**-Befehl bricht, sobald er erreicht wird, eine Schleife (oder einen Case) ab.
return; bzw. **return**<Datentyp>;	Siehe Funktionen/Methoden.
switch(<Variable>) {**case** <Variablenwert>: ... **break**; ... **default**:...}	Die Fallunterscheidung **switch**ähnelt einer **if-else**-Entscheidung mit vielen Fällen, wobei nur der Wert einer Variable betrachtet wird. Es werden verschiedene Fälle (**cases**) geprüft und dementsprechend der Code bis zur nächsten **break**-Anweisung ausgeführt (Aufpassen! Fehleranfällig!). Der **else**-Fall enspricht dem **default**-Case, der optional am Ende der Fallunterscheidung stehen kann. Hier ist kein **break;** erforderlich. Möchte man bei mehreren Fällen den selben Code ausführen, dann kann man die Cases hintereinander vor dem Code angeben (**case** <Wertert1>: **case** <Wertert2>: **case** <Wertert2>: ... **break;**).
this.<Variable>	Gibt es Überschneidungen von Parameternamen und Klassenvariablen, kann durch „**this**.<Variable>“auf die Klassenvariable und mit <Variable> auf den überreichten Parameter zugegriffen werden. Hauptsächlich wird das bei Settern verwendet, kann aber auch generell immer bei Klassenvariablen genutzt werden.
continue;	Der **break**-Befehl überspringt einen Schleifendurchlauf und springt zurück zum Schleifenheader (Die Veränderung wird ausgeführt und die Abbruchbedingung wird geprüft).

enum	Ein Enum ist eine spezielle Klassenform. Sie wird als Sammlung von Konstanten verwendet. Eine solche Konstante besteht aus einem String und einer Nummer. Beispiel: „**enum** `BloodType {A, B, ZERO};`“erstellt ein Enum mit drei Konstanten. Dabei haben sie die Werte (`Ä", 0`), (`"B", 1`) und (`SZERO", 2`). Der Zugriff auf den String erfolgt durch die `.toString()`-Methode, auf den Zahlenwert mit der `.ordinal()`-Methode der jeweiligen Konstante.
public, **protected**, **private** (oder keiner davon)	Ein Modifier der angibt, von wo man auf eine Variable/Funktion/(Klasse) im Projekt zugreifen kann. Wird **private** verwendet, ist die Variable nur in der Klasse selbst erreichbar/sichtbar; Ohne Modifier kann man von Klassen im gleichem Package darauf zugreifen; Mit **protected** sind sie zusätzlich vom Package auch in allen Subklassen sichtbar. **public** gibt an, dass die Variable von überall erreichbar ist.
static	(Vor der Objektorientierten Programmierung sollte **static** für alle Funktionen und Klassenvariablen verwendet werden). Der Modifier **static** gibt an, dass die Variable/Funktion schon bei Programmstart existiert und somit keine Instanzierung zur Verwendung nötig ist.
final	**final**macht eine Variable zur Konstante. Sie können nach der Initalisierung nicht mehr verändert werden.
class	Steht für einen „Klassenblock“dar, in dem sich alle Funktionen und Variablen befinden.
package	Steht außerhalb des „Klassenblock“und gibt das Package der sich die Klasse befindet.
import	Steht außerhalb des „Klassenblock“und gibt alle anderen Klassen (Funktionen und Werte), die in dieser Klasse verwendet werden.
main	Methodenname der für die Main-Methode reserviert ist.
extends	„Erweitert“die (Eltern-)klasse um die angegebenen aktuelle Klasse(Kind). Die aktuelle Klasse kann nur eine einzige Klasse erweitern. Entsprechende Funktionen der Elternklasse werden übernommen und können bei Bedarf überschrieben werden.
super.`<Funktion>`	Zugriff in einer Kindsklasse auf die überschriebene Funktion der Elternklasse.
abstract	Eine abstrakte Klasse kann nicht instanziert werden. Sie kann einige abstrakte Funktionen besitzen, die nicht implementiert werden müssen sondern in jeder Kindsklassen.

`interface`	Steht für ein Interface. Grob gesagt, ist das eine Angabe der Methoden und Variablen in einer Klasse. Wird ein Interface verwendet (implementiert) MÜSSEN die Funktionen in dieser Klasse implementiert werden.
`implements`	Die Klasse implementiert ein Interface.
`<Labelname>:` und `break<Labelname>;`	Möchte man aus mehreren, verschachtelten Schleifen herausspringen kann man ein Label mit „`<Labelname>:`“definieren. Dieses wird der direkt darauffolgenden Schleife zugeordnet. Möchte man dann diese Schleife beenden, geschieht das mit „`break<Labelname>;`“

Nützliche Shortcuts

Strg + LEER	Autovervollständigung. Der Text „`sysout`“wird zum Befehl `System.out.println();` ergänzt.
Strg + I	Einzüge der Auswahl/Zeile korrigieren.
Strg + Shift + F	Autoformatierung des Codes (Einzüge, etc.).
Strg + 7	Kommentiert die aktuelle Zeile/Auswahl aus/ein.
Strg + S; Strg + Shift + S	Klasse speichern; Alles Speichern.
Strg + (Mausklick auf Variable/Methode)	Springt zur Deklaration der Variable/Methode.
Strg + .	Springt zum nächsten Fehler (gelb oder rot unterkringelt).
Strg + Shift + O	Räumt Imports auf.
Strg + F11	Programm ausführen.
Strg + Shift + L	Anzeigen aller Shortcuts.
Strg + C; Strg + V; Strg + X; Strg + A	Kopieren; Einfügen; Ausschneiden; Alles auswählen.
Strg + Z; Strg + Y	Befehl rückgängig machen; Befehl wiederholen.
(Source>Generate Getters and Setters...)	Dort kann man Auswählen zu welchen Variablen Getter und Setter erstellt werden sollen.
(Auswahl eines Projekts/-Pakets/Klasse, Rechtsklick, Refactor>Rename...;)	Umbenennung einer des Projekts, Pakets oder der Klasse und Aktualisierung im gesamten Projekt.