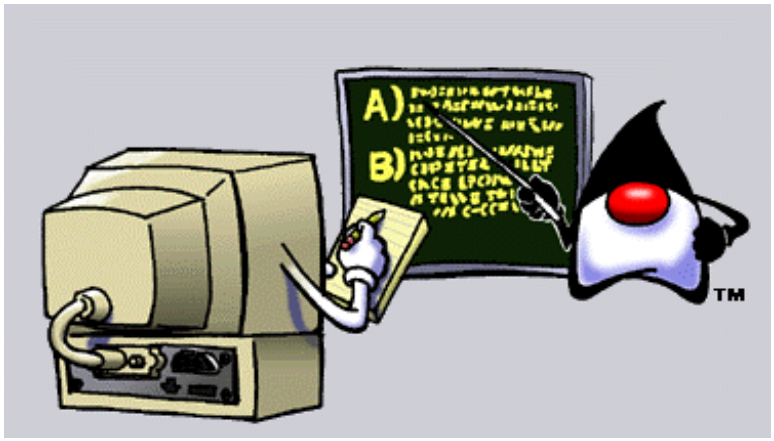
In diesem Kapitel

Java Programmierung - eine Einführung

2.1. Über diesen einführenden Teil des Kurses

Dies ist der zweite Kurs, welcher Sie in die Rechnergrundlagen einführen soll und Ihnen die Flexibilität und die Möglichkeiten der Java™ Programmiersprache näher bringen.

Dieser Kurs wurde für Personen entwickelt, welche über limitiertes Wissen über Rechner und die Programmierung verfügen. Damit Sie diesen Kurs erfolgreich abschliessen können, sollten Sie den ersten Kurs, *Rechner- und Programmier- Grundlagen* besucht haben oder folgende Kenntnisse haben:

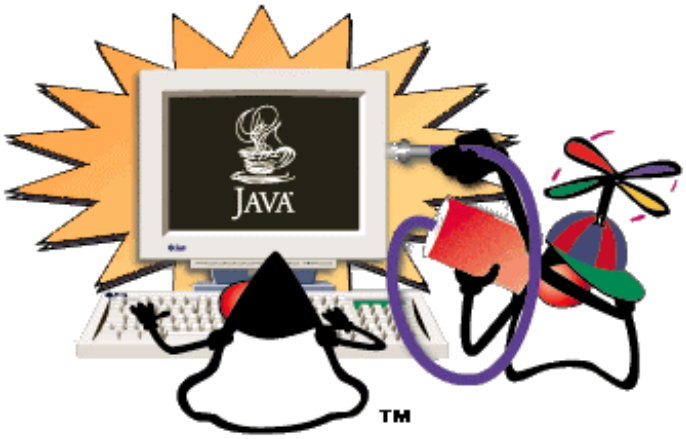


- grundlegende Prinzipien eines Rechners
- die Standardkomponenten eines Rechners
- elementare Programmierkenntnisse
- grundlegende Prinzipien der Software- und Produktentwicklung
- einfache Grundkenntnisse der objektorientierten Konzepte
- elementares Hintergrundwissen zu Java

2.2. Kurs Einführung

Herzlich willkommen zum Kurs.

Als Kurzinformation, erinnern Sie sich:
Java Technologie und Programme bestehen aus Klassen, aus denen Objekte gebaut werden. Das Verhalten dieser Objekte wird durch Methoden beschrieben und die individuellen Charakteristiken, welche Objekte voneinander unterscheiden, werden Attribute genannt.



2.2.1. Klassen

Klassen sind Templates, Platzhalter oder Vorlagen, aus denen Objekte gebildet werden. Klassen modellieren Ideen und Konzepte innerhalb des Problembereichs, der mit einem Programm gelöst werden soll. Vereinfacht gesagt wird es also so viele Klassen wie Konzepte innerhalb des Problembereichs geben. Klassen sollten nach Möglichkeit mir realen Konzepten korrespondieren. Beispiele wären also: eine Katze, eine Türe, usw. Klassen, die Sie definieren, hängen also von Ihrer Erfahrungswelt ab und dem Problem, welches Sie lösen möchten.

Eine Klasse ist ein Konstrukt, eine Programmierstruktur, mit deren Hilfe Daten und Funktionen (objektorientiert "Methoden" genannt) zusammengefasst werden und die in Form von Programmcode spezifiziert werden.

2.2.2. Objekte

Objekte sind Entitäten, welche gebildet werden, wenn Sie Ihr Programm starten. In Objekten werden die Daten gespeichert und manipuliert, welche benötigt werden, um Ihr Problem zu lösen. Die Daten werden mit Hilfe von Methoden, den Methoden des Objekts (welche in der Klasse definiert werden), manipuliert.

2.2.3. Methoden

Die Funktionen oder das Verhalten, welches ein Objekt zeigt, werden objektorientiert als Methoden bezeichnet. Methoden bestehen aus Anweisungen, welche bestimmte Aufgaben einer Klasse erfüllen.

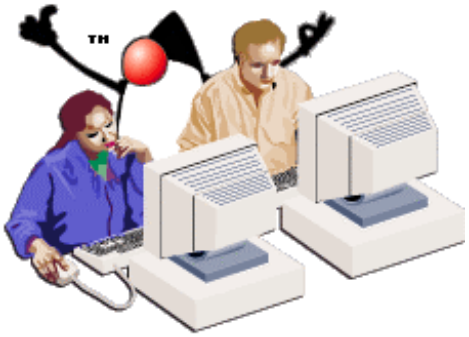
Beispielsweise könnte eine Methode eines Bankkontos dafür zu sorgen haben, dass der Kontostand abgefragt wird, also der Kontostand an einen externen Beobachter des Kontos übergeben wird. Die Aufgabe "abfragenKontostand()" ist eine Aufgabe, welche auf Stufe Bankkonto, also der Klasse, definiert werden.

2.2.4. Attribute

Attribute definieren / beschreiben ein Objekt. Attribute werden in Variablen innerhalb eines Objekts beschrieben. Jedesmal, wenn ein Objekt gebaut wird, werden (sollten) die Variablen initialisiert werden, also mit Werten belegt werden.

Beispielsweise in der Klasse Katze: bei der Bildung eines Objekts, welches zur Klasse Katze gehört (ein Katzenobjekt, eine konkrete Katze, eine Instanz der Klasse Katze), sollte der Name der Katze angegeben werden. Natürlich haben oder können unterschiedliche Katzen (Objekte) unterschiedliche Namen haben ("Stinky", "Peterli", "Käthi").

2.2.5. Kursziele



Nach dem Durcharbeiten dieses Kurses sollten Sie in der Lage sein:

- die Regeln und Werkzeuge der Java Programmiersprache einzusetzen:
  schreiben von Programmen
  übersetzen von Programmen
  starten einer kleinen Applikation

- Java Konstrukte einzusetzen, mit denen Sie Programme erweitern können.

- Programme zu schreiben, welche komplexere Kontrollstrukturen (bedingte Ausführung, Schleifen) benutzen

- Kapselungen einzusetzen, um Daten vor externen Einflüssen zu schützen.

- Arrays einzusetzen, um effizient mit Schleifen arbeiten zu können (Algorithmen [Schleife] und Datenstrukturen [Arrays] hängen voneinander ab).

- objektorientierte Werkzeuge einzusetzen, um Applikationsprogramme zu entwickeln.

2.3. Module 1: Java Programmiersprache und Werkzeuge

2.3.1. Einführung in diesen Modul

In diesem Modul lernen Sie die Regeln kennen, die Sie verstehen müssen, und die Werkzeuge, die Sie kennen müssen, bevor Sie mit dem Entwickeln von Java programmen beginnen können.

2.3.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- die Regeln und Werkzeuge der Java Programmiersprache anzuwenden
- Deklarationen und Anweisungen zu schreiben
- die Datentypen von Java identifizieren zu können
- die Wertebereiche der Java Datentypen zu kennen
- kleine Java Applikationen zu übersetzen und zu starten.

2.3.1.2. Fragen an sich selbst

Bereiten Sie sich für das Durcharbeiten dieses Moduls vor, indem Sie sich folgende Fragen stellen.

- wie kann ich Werte speichern, die ich in meinem Programm benötige?
- welche Arten von Werten kann ich speichern?
- sind Objekte die einzigen Variablen, die ich benutzen kann?

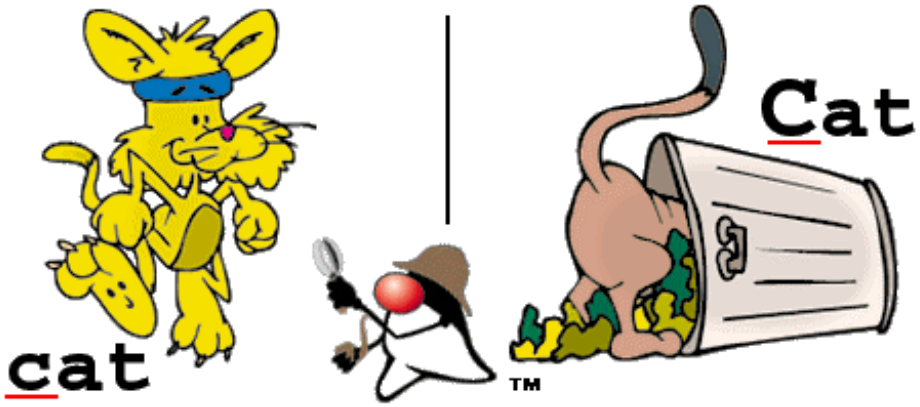
Aktivieren Sie Ihr Wissen! Indem Sie sich Gedanken machen zu den obigen Fragen, sollten Sie erkennen, ob es für Sie überhaupt Sinn macht, diesen Modul durchzuarbeiten oder ob Sie gleich zum nächsten wechseln sollten.

Falls Sie fortfahren sollten Sie die Fragen im Kopf behalten. Sie helfen Ihnen beim Verständnis des Textes.

2.3.2. Programmier Regeln und Werkzeuge

2.3.2.1. Worte sind Case sensitiv

Case sensitiv bezieht sich auf die Darstellung oder Schreibweise in Grossbuchstaben oder Kleinbuchstaben.
Java Programmcode besteht aus Worten und Werten, wie jeder Programmcode. Aber der Compiler versteht nur eine begrenzte Anzahl Worte - die ***Schlüsselworte.*** Daher muss der Programmierer darauf achten, dass er diese Schlüsselworte korrekt einsetzt. Alle Schlüsselworte sind ausschliesslich in Kleinbuchstaben geschrieben
Beispielsweise ist eines der Schlüsselworte `class`. Der Compiler versteht weder `Class` noch `CLASS`. Case sensitiv bezieht sich auch auf die Variablennamen.



Beispielsweise könnten Sie eine Variable `cat` kreieren. Sie können dann nicht plötzlich in Ihrem Programm die Katze als `Cat` ansprechen.

2.3.2.2. Kommentare

Weil der Java Übersetzer nur Schlüsselworte und Programmcode versteht, muss alle andere in Form von ***Kommentaren*** im Programm stehen, damit der Compiler diese Teile ignoriert.
Informationen, die vom Compiler ignoriert werden sollen, aber vom Benutzer gelesen werden können, beinhalten:
Beschreibungen der Konzepte, Erklärungen für bestimmte Aktionen des Programms,
Metadaten,wie beispielsweise wann die Daten das letzte Mal gesichert wurden.
Der Compiler ignoriert alles, was durch bestimmte Zeichenkombinationen eingefasst wird:

Einzeilige Kommentare - ein // teilt dem Compiler mit, dass alles bis zum Ende der aktuellen Zeile ignoriert werden soll.
Beispiel:

```
//Dies ist ein einzeiliger Kommentar
```

Mehrzeiliger Kommentar - die Zeichenkombination /* teilt dem Compiler mit, dass alle Zeilen bis zum Kommentarabschluss */ ignoriert werden sollen.
Beispiel:

```
/* Dieser Kommentar geht über mehr als
eine Zeile. */
```

JavaDoc Kommentar - die Zeichenkombination /** teilt dem Compiler mit, dass alle Zeilen bis zum Kommentarabschluss */ ignoriert werden sollen und zu JavaDoc gehören
Beispiel:

```
/**
ein JavaDoc Kommentar
*/
```

2.3.2.3. Schlüsselworte

Alle Programmiersprachen besitzen Schlüsselworte. Schlüsselworte haben eine spezielle Bedeutung für den Compiler, falls sie in der richtigen Reihenfolge (***Syntax***) geschrieben werden. Falls die Reihenfolge nicht stimmt, werden Fehlermeldungen generiert.

Die Schlüsselwörter werden benutzt, um bestimmte Aktionen auszuführen oder Objekte in einer vordefinierten Art und Weise zu behandeln. In der Regel gibt es auch Worte, welche nicht erlaubt sind. Das kennen wir auch aus der Umgangssprache: ein Kind "Dieses" zu taufen, wäre nicht nur für das Kind selbst etwas irritierend; man könnte sich darunter schlicht nichts vorstellen.

Schlüsselworte dürfen nicht eingesetzt werden für:
Klassennamen, Methodennamen, Variablennamen und so weiter.

Schlüsselworte können eingesetzt werden, um eine Variable (`int integerVariable`) zu definieren, oder um eine Aktion anzuzeigen (`extends`).

In der Tabelle unten sind die Schlüsselworte der Java Programmiersprache aufgeführt (inklusive der reservierten Worte, welche nicht benutzt werden, in Schrägschrift).

Reservierte Worte sind Schlüsselworte, welche von der Programmiersprache *reserviert*, aber aktuell nicht implementiert sind. Die Schlüsselworte, die in dieser Kurseinheit behandelt werden, sind **fett** geschrieben.

Beispiele: *goto* ist reserviert; **abstract** werden wir besprechen. Andere Schlüsselworte (this, synchronized, ...) folgen in einer der folgenden Kurseinheiten.

abstract	**default**	*goto*	**null**	synchronized
boolean	**do**	**if**	package	this
break	double	implements	**private**	throw
byte	**else**	import	protected	throws
case	**extends**	instanceof	**public**	transient
catch	**false***	**int**	return	**true***
char	final	interface	**short**	try
class	finally	**long**	static	void
const	**float**	native	super	volatile
continue	**for**	**new**	**switch**	**while**

* `true` und `false` sind zwar aufgeführt, sind aber strikt gesprochen Literale (Werte von Variablen) und keine Schlüsselworte. In Java werden die Wahrheitswerte klein, in C gross geschrieben: in Java `true, false`; in C `TRUE, FALSE`.

2.3.2.4. Layout der Programme (Code Layout)

In Java verwendet man bestimmte Codesymbole und Layoutkonventionen, um Programme lesbarer zu machen.

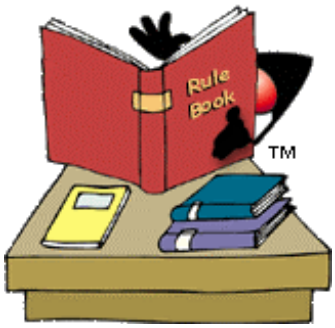
Anweisungen - eine Anweisung ist Code, der eine bestimmte einfache Operation ausführt - wie beispielsweise eine Zuweisung (assignment) eines Wertes an eine Variable oder die Ausgabe auf einen Drucker.

Anweisungen müssen mit einem Semikolon (;) abgeschlossen werden

```
i = 6 + 9;
```

Ausdruck - ist eine Anweisung, welche einen Wert zurück liefert, wie beispielsweise in einer mathematischen Berechnung.

Codeblöcke - oft benötigt man mehr als eine Anweisung, um eine Aufgabe zu erfüllen. In diesem Fall werden alle Anweisungen, die zusammen die Aufgabe lösen, zusammengefasst und mit Klammern begrenzt werden - { ...// Codeblock ....} // Ende des Codeblocks
Syntaktisch ist ein Codeblock äquivalent zu einer einzelnen Anweisung. Das heisst, dass man überall dort, wo man eine einfache Anweisung verwenden kann, auch einen Codeblock verwenden darf.




```
{i = 6 + 9; j = i - 3;}
```

Klammern werden auch benutzt um Klassen und Methoden einzufassen:


```
class Baseball { // Klassendefinition
  // Anweisungen
}
```

Whitespace - Whitespace beschreibt Zeichen im Programmcode, welche keinen Einfluss auf die Ausführung des Programms haben und lediglich die Lesbarkeit des Programms erhöhen. Da Java eine frei-formatige Programmiersprache Sprache ist, können Sie Programmzeilen auf eine Zeile, oder auch auf mehrere Zeilen verteilt schreiben. Dadurch wird das Programm aber nicht lesbarer und wartbarer. Die Klammern werden in strukturierten Programmen einrückt, wie man das üblicherweise tun sollte.

```
{
     i = 6 + 9;
     j = i - 3;
}
```

2.3.2.5. Identifiers (Bezeichner)

Um Daten in einem Programm zu speichern, muss man die Daten mit einem Bezeichner verknüpfen. Ein Bezeichner ist schlicht ein Name für diese Grösse. Dieser Name entspricht einem "Speicherbereich", in dem die Werte des Bezeichners abgespeichert sind.

Jede Klasse, jede Methode und jede Variable besitzt einen Bezeichner. Die folgenden regeln bestimmen den Inhalt und die Struktur eines Bezeichners:

- das erste Zeichen eines Bezeichners, eines Namens, muss sein, entweder:
  - ein Grossbuchstabe (A-Z)
  - ein Kleinbuchstabe (a-z)
  - ein underscore Zeichen (_)
  - ein Dollarzeichen ($)
- das zweite und die weiteren Zeichen eines Namens muss sein:
  - irgend ein Zeichen aus der obigen Liste
  - ein nummerisches Zeichen (0-9)

Es ist *unter keinen Umständen* gestattet, eines der Java Schlüsselworte als Namen, Bezeichner, zu verwenden.

Zeichen anderer Sprachen (englisch, französisch, italienisch,...) können in Namen auch benutzt werden, beispielsweise Umlaute (ä,ö,ü; è,é,à,...).

Allerdings sollte man der Lesbarkeit zu liebe darauf verzichten.

2.3.2.6. Variablen und Konstanten

Variablen und Konstanten sind in "Speicherbereichen", in denen Werte gespeichert werden können.

- **Variablen**
  Der Wert einer Variablen kann im Verlaufe der Zeit, der Programmausführung verändert werden. Es gibt drei Arten von Variablen:
  1. ***Klassenvariablen***
     speichert Informationen über die Klasse selbst. Beispielsweise könnte die letzte vergebene Serienummer in der Klasse abgespeichert werden.
  2. ***Instanzenvariablen***
     speichern Informationen über einzelne Objekte oder Instanzen der Klasse (Objekte = Instanzen einer Klasse). Sie möchten beispielsweise Informationen über eine bestimmte Katze abspeichern.
  3. ***lokale Variablen***
     werden innerhalb einer Methodendefinition oder eines Blocks, einer Schleife benötigt. Diese Variablen existieren nur temporär, solange wie das Programm sie innerhalb deren Gültigkeitsbereich verwendet. Anschliessend ist diese Variable nicht mehr definiert. Die Variable `sterne` in folgendem Beispiel wird lediglich des Blocks definiert. Ausserhalb des Blocks existiert die Variable nicht mehr:

```
{
int stars;
stars = x + 2:
//...
}
```

- **Konstanten**
  In anderen Speicherbereichen werden die Variablen aus dem Programm abgespeichert, welche ihren Wert nicht ändern können, also Konstanten. Beispielsweise, falls Sie im Programm die Konstante "pi" benötigen und nicht die Java Version dafür verwenden möchten, dann macht es Sinn eine Konstante "pi" zu definieren und nicht mehr zu verändern.

2.3.2.7. Namenskonventionen

Sun, aber auch andere Firmen und Organisationen, haben einige einfache Regeln und Konventionen aufgestellt, um das Aussehen (Look & Feel) des Java Programmcodes zu standardisieren. Diese Standards berücksichtigen, dass ein Bezeichner eine ununterbrochene (zusammenhängende) Zeichenfolge sein muss, welche mehrere Worte umfassen kann, beispielsweise `meineKatze` (bestehend aus den zwei Worten "meine" und "Katze") als Objektname, um die Bedeutung klarer ausdrücken zu können.

- **Klasse**
  Klassenamen werden als eine *Sequenz von beschreibenden Worten* (in der Regel Nomen, Dingwörter, Substantive und Adjektive), wobei jedes Wort mit dem nächsten verknüpft wird und jedes Wort mit einem Grossbuchstaben startet und mit Kleinbuchstaben weiterfährt.

```
class ClassOne {}
class MyPetRhinoceros {}
```

- **Variable**
  Diese Namen folgen der selben Regel wie die Klassennamen, müssen aber mit einem *Kleinbuchstaben* beginnen:

```
int variableOne;
String myFirstReferenceVariable;
```

- **Konstante**
  Diese Namen verwenden lediglich *Grossbuchstaben* für alle Worte und underscore als Trennzeichen für die Worte:

```
final int CONSTANT_ONE = 27
final double PI = 3.141592654
```

- **Methoden**
  Die Methodennamen gehorchen demselben Schema wie Variablen mit der Ausnahme, dass Methodennamen normalerweise mit einem *Verb* beginnen und mit Klammern enden. Die Klammern sind nict Teil des Namens sondern des Methodenaufruf Mechanismus.

```
int methodOne() {}
void printTheResults() {}
```

2.3.3. Programmier Regeln und Werkzeuge - Review

Vervollständigen Sie folgende Sätze:[1]

Ein ..... ist der Programmcode, welcher eine einfache Operation ausführt - beispielsweise eine Wertzuweisung an eine Variable oder die Ausgabe auf einen Drucker.

Statement (Anweisung)
Ausdruck
Programmblock
Whitespace

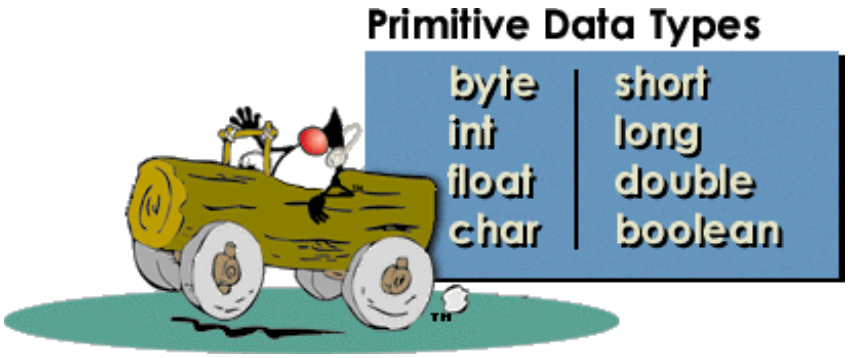
[1] Statement

2.3.4. Datentypen, Variablentypen

Zur Wiederholung: Variablen sind Speicherlokationen, in denen Daten gespeichert werden können. Da "Daten" unterschiedliche Grössen sein können, ein Wort, eine Zahl oder sonst irgend etwas, wie beispielsweise "true", ist der Speicherbedarf für die Speicherung der Daten unterschiedlich gross.

Um die Effizienz zu steigern teilt man Variablen und Konstanten in drei Datentypen ein:

1. **Primitive (Basisdatentypen)**
   primitive Datentypen oder Basisdatentypen werden vom System zur Verfügung gestellt Sie stellen die Basisbausteine dar für den Bau aller anderen Datentypen. Variablen, welche als primitive Datentypen deklariert werden, sind keine Objekte.



   In der Grafik oben sehen Sie die acht Basisdatentypen, welche wir im Folgenden in diesem Modul besprechen werden. Falls Sie einen der Basisdatentypen einsetzen, weiss das System genau wieviel Speicherplatz dafür benötigt wird und in welchem Speicherbereich diese Variablen gespeichert werden.

2. **Referenzen**
   Referenzvariablen werden benutzt um die Adressen von Objekten, die man kreiert, zu speichern. Wenn man ein neues Objekt kreiert, weiss das System nicht, wieviel Speicherplatz für das Speichern des Objekts benötigt wird. Das System muss sich die Speicheradresse des Objekts merken.

3. **Array, Datenfelder**
   Ein Array ist eine spezielle Variable, welche mehrere Objekte oder Basisdatentypen aufnehmen kann.

2.3.4.1. Der Basisdatentyp Integer

Der Java Programmiersprache Compiler prüft die Zuweisungen zwischen Variablen und Objekten, um sicherzustellen, dass sie vom selben Datentyp sind, oder dass die Zuweisung zwischen den Datentypen den Syntaxregeln (Compilerregeln) genügt.

Integers sind die ganzen Zahlen -1, 1, 2, 3, 4, usw. Java kennt vier Integer Datentypen, welche mit den Schlüsselworten `byte, short, int` und `long` bezeichnet werden. Der Datenbereich (die maximale Anzahl möglicher Werte) wird bestimmt durch die Anzahl Bits, mit denen der Wert gespeichert wird.

- jeder Integer, der ganze Bytes enthält - ein `byte,` ist acht Bits lang.
  jeder Integer vom Typ `short` ist zwei Byte, also 16 Bits lang.
  jeder Integer vom Typ `int` ist vier Bytes, also 32 Bits lang.
  jeder Integer vom Typ `long` ist acht Bytes, also 64 Bits lang.

  Damit kann eine `short` Integerzahl doppelt so lange Zahlen darstellen, wie eine `byte` Integerzahl; eine `int` kann Zahlen darstellen, die zweimal so lang sind wie `short` und so weiter.

- ganze Zahlen können entweder positiv oder negativ sein.

- das Vorzeichen eines Wertes wird mit dem Bit ganz links dargestellt.
  Eine "1" an dieser Stelle bedeutet, dass es sich um eine negative Zahl handelt.
  Eine "0" an dieser Stelle bedeutet, dass es sich um eine positive Zahl handelt.

  In einer Variable vom Datentyp `byte,` bei dem ein Bit für das Vorzeichen verwendet wird und die restlichen sieben Bits eine Zahl darstellen können,
  Jedes der sieben Bits kann entweder 0 oder 1 sein. Daher kann man 2 hoch 7 Werte darstellen, oder -128 ...127 da die 0 als ein positiver Wert bewertet wird.

- ein `short` Integer enthält 16 Bits, kann also 2 hoch 15 Werte darstellen:
  -32768 bis +32767.

Falls wir also eine Variable hätten, welche maximal 100 werden kann, dann reichen 8 Bits oder ein Byte, also eine Variable vom Datentyp `byte`, um alle Werte darzustellen.
Und hier je ein Beispiel für die Deklaration von Integer Datentypen:
die korrekte Syntax ist: <Type> <Bezeichner> ; (Semikolon).

```
byte num8;
short num16;
int num32;
long num64;
int iVar;
int i1, i2, i3;
```

(die Zahlen hinter dem oberen Variablennamen ist unwichtig und hier nur zur Erinnerung an die Anzahl Bits hingeschrieben).

2.3.4.2. Der Basisdatentyp Floating Point

Floating Point (Fliesskomma) Zahlen, manchmal auch (*engl.*) *real* Zahlen, enthalten einen Dezimalpunkt.

Es gibt zwei Typen von Fliesskommazahlen, `float` und `double`. Beide Typen sind funktional gleichwertig aber unterschiedlich lang.

Float Länge	**Name oder Typ**
32 bits	float
64 bits	double

Bei Fliesskomma Zahlen lassen sich die Datenbereiche nicht so einfach angeben wie beim Integer Datentyp. In Java hat man zwei Konstanten (also gross geschrieben, mit einem "_" als Verbindung zwischen zwei Worten, gemäss der Namenskonvention in Java), definiert

- POSITIVE_INFINITY
- NEGATIVE_INFINITY

Selbsttestaufgabe 1

Versuchen Sie herauszufinden, wie gross diese beiden Konstanten sind.

Um eine Fliesskommavariable einzusetzen wird der Variablentyp plus Variblenname in eine Anweisung zusammengefasst:

```
double myFirstFloatingPoint;
```

Nach der Deklaration der Variable kann man diese einsetzen, um Werte zu speichern. Dies geschieht mit Hilfe einer Zuweisung (in einer Anweisung; *engl. assignment in a statement*):

```
myFirstFloatingPoint = 27.999;
```

Für grosse Zahlen kann man auch die "Engineering" Notation benutzen, bei der man mit Zehnerpotenzen arbeitet und diese hinter ein "E" oder "e" schreibt:

```
myFirstFloatingPoint = 2.7999E10;
```

2.3.4.3. Der Basisdatentyp Zeichen - `char`

Wir wollen auch Zeichen und Zeichenketten (Wörter) abspeichern.
Für die Speicherung von einzelnen Zeichen benötigen wir den Basisdatentyp `char`.

In einer Variable vom Typ `char`, kann man lediglich ein Zeichen abspeichern. Ganze Worte werden in Variablen vom Basisdatentyp `String` abgespeichert.

Eine Variable zum Speichern eines Zeichens wird folgendermassen deklariert:

```
char myFirstCharacter;
```

Werte werden durch eine Anweisung einer solchen Variable zugewiesen. Der Wert,den Sie speichern wollen, muss ein einfaches Zeichen sein und muss in Anführungszeichen (') eingeschlossen werden. Die Anführungszeichen konvertieren das einzelne Zeichen beispielsweise `g` in einen Character / Zeichenwert mit dem Wert '`g`':

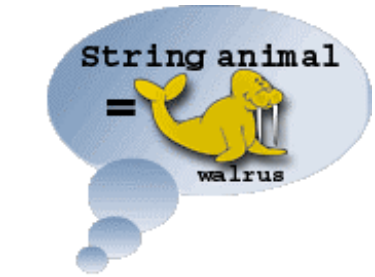
```
myFirstCharacter = 'g';
```

Variablen vom `char` Typ können beliebige Werte des Java Zeichensatzes speichern, also auch japanische oder chinesische Zeichen. Die meisten Programmiersprachen verwenden den American Standard Code for Information Interchange (ASCII), ein 8-bit Zeichensatz, mit dem alle Zeichen dargestellt werden können, die im Englischen vorkommen.

In Java wird eine andere (international standarisierte) 16-Bit Codierung verwendet, mit der man so gut wie alle Zeichen, die weltweit irgendwo im Einsatz sind, darstellen kann: Unicode. (Sie finden viele Details zu diesem Code unter http://ww.unicode.org)

2.3.4.4. Der Basisdatentyp Zeichenkette – `String`

Sie können ein einzelnes Zeichen in einer `char` Variablen abspeichern. Aber falls Sie ganze Wörter, mehrere Wörter oder ganze Sätze abspeichern möchten, benötigen Sie die `String` Klasse. Hier handelt es sich also bereits um eine *Klasse*, im Gegensatz zu den früher besprochenen Datentypen (Strings kann man aus `char` zusammensetzen).



Zeichenketten sind eine spezielle Klasse, welche ganze Serien von Zeichen abspeichert.

Die `String` Klasse existiert als eine von vielen Standardklassen, welche mit dem Java Development Kit (JDK™) ausgeliefert wird.

Im nebenstehenden Bild wird die Deklaration einer Zeichenkette illustriert. Hier ist diese Deklaration in Java:

```
String animal = "walrus";
```

Da Zeichenketten eine Klasse bilden, handelt es sich hier eigentlich bereits um ein Objekt, das Objekt `animal` mit dem Wert "`walrus`" ist ein Objekt zur Klasse `String` (eine Instanz der Klasse `String`).

2.3.4.5. Der Basisdatentyp `boolean`

In vielen Programmen muss man die Möglichkeit vorsehen, zu verzweigen, je nach Wert einer Variablen. Diese Entscheide basieren häufig auf Berechnungen, welche als Ergebnis einen Wahrheitswert liefern (trifft zu, trifft nicht zu; wahr falsch; "true", "false").

Falls wir solche Entscheide speichern wollen, benötigen wir einen Datenspeicher für logische, `boolean` Variable.

Boole'sche Variable gehorchen denselben Namenskonventionen, wie andere Variablen in Java. Aber im Gegensatz zu den andern Basisdatentypen (`int, float`, ...) besitzen sie lediglich zwei Werte: `true` und `false`.

Alle Variablen, welche zweiwertige Zustandskodierungen (ein, aus, "on", "off") beschreiben, können auf Boole'sche Werte abgebildet werden.

Und hier ein Beispiel für die Deklaration (Vereinbarung) und Zuweisungen (engl. *assignments*) von `boolean` Variablen:

```
// Stundenlohn oder Gehalt ?
boolean     is_salaried;
boolean     is_hourly;

is salaried = true; // Wert der Variable is_salaried ist true
is hourly = false;
```

Bemerkung: in einem realistischen Programm würden Sie nur eine der Variablen verwenden und den andern Zustand als Standardwert annehmen. Hier würde man also annehmen, dass alle ein Gehalt haben (meine Annahme), dass es aber Fälle gibt, bei denen ein Mitarbeiter pro Stunde bezahlt wird.

2.3.4.6. Referenzvariablen und das `new` Keyword

Grundsätzlich werden Klassen und Objekte auf folgende Art und Weise gebaut:

1. Definition der Klasse: <Namen>, <Attribute>, <Methoden>
   Beispiel:
   Name der Klasse: Angestellter (`employee`)
   Attribute : Personalnummer, Gehalt, ... (`EmpNum, is_salaried, Salary,...`)

   ```
   class Employee {
      int empNum;
      boolean is_salaried;
      double salary;
      ...
   }
   ```

2. Nun haben wir ein Template, eine Vorlage, aus der hervorgeht, was (welche Informationen) wir abspeichern möchten oder auch, auf welche Informationen wir zugreifen möchten (damit wir dies können, müssen wir sie irgendwo erfassen).

   Pro Mitarbeiter unsere Organisation werden wir ein Objekt als Instanz dieser Klasse kreieren und in diesem Objekt die Daten des Mitarbeiters abspeichern.

   Aber wie können wir die einzelnen Objekte, die einzelnen Mitarbeiter voneinander unterscheiden?
   Indem wir jedem einen Namen geben, unter dem wir ihn ansprechen können.

   Beispiel:
   ```
   Employee Jens;       // definiert (nicht kreiert) den Mitarbeiter Jens
   ```

   Nun können wir wann immer wir ihn brauchen, auf die Daten vom Mitarbeiter Jens zugreifen. Aber die Attribute, die Datenfelder von Jens sind noch nicht mit Werten gefüllt!

Was wir nach der Definition der Klasse und der Deklaration des Objekts noch zu tun haben ist, das Objekt auch zu kreieren! Die Deklaration besagt lediglich, dass wir gerne ein Objekt hätten der entsprechenden Klasse. Das System legt aber noch keinen Speicherbereich für das Objekt an; es reserviert höchstens Platz, weil vermutlich Speicherplatz benötigt wird.

In Java werden neue Objekte mit einer speziellen Methode (Funktion), dem ***Konstruktor*** kreiert und initialisiert. Das Schlüsselwort `new` zeigt an, dass ein neues Objekt kreiert wird. Die folgende Anweisung kreiert (oder konstruiert) Jens:

```
Jens = new Employee( );
```

Wie Sie auf Grund der öffnenden und schliessenden Klammer sehen, handelt es sich um einen Methodenaufruf.

Was passiert konkret?
Das Template `Employee` wird genommen und im Speicher abgelegt. Diesen Speicherbereich können wir über den Namen `Jens` ansprechen.

Die Zuweisung über das Zuweisungs- Symbol = ordnet Werte einer Variable zu. In unserem Fall werden alle Attribute und Methoden, welche zusammen die Definition der Klasse `Employee` ausmachen `Jens` zugeordnet. Nun weiss das System auch wieviel Speicherplatz für `Jens` reserviert werden muss.

Was wir noch machen können (wohl auch müssen, denn sonst nützt das Objekt kaum etwas), ist das Objekt mit Informationen über Jens zu füllen:

```
Jens.empNum = 1001001;        // Jens Personalnummer
Jens.is_Salaried = true;      // auch Jens kriegt ein Gehalt
Jens.salary = 65000;          // Anfangsgehalt eines Inf. Ing.
```

2.3.5. Datentypen - Review

Hier einige Frage, die Ihr Wissen prüfen sollen:

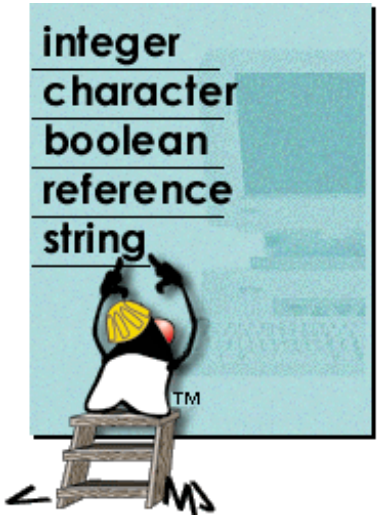
1. Es gibt zwei Arten von Fliesskommazahlen: ..... und .....

   mögliche Antworten:[2]
   int & short
   float & double
   long & char
   char & boolean

[2] float & double

2.3.6. Literale Werte

Die verschiedenen Datentypen (Datenarten), welche wir bisher betrachtet haben, besitzen Variablen mit **literale** *Werte* (Werte wurden ihnen zugeteilt wurden) und welche vom Compiler direkt verstanden werden können. Der Compiler schickt diese im Programm fest vergebenen Werte an die JVM und teilt ihr mit, dass es sich dabei um "Konstanten" ohne Namen handelt.



Da die Variable namenlos gespeichert wird, muss sie in einem bestimmten, für solche Datentypen reservierten Speicherbereich stehen. Aus der Lokation kann man dann herleiten, welche Operationen mit diesen Daten ausgeführt werden können. (Daten im Integerbereich können Integeroperationen durchführen, Daten im char-Bereich können Zeichenoperationen durchführen bzw. man kann Zeichenoperationen mit ihnen ausführen usw.).

Literal heissen die Werte, weil man sie explizit eintippt und als Text sieht. Im Gegensatz dazu gibt es andere Werte, die in komplexen Berechnungen ermittelt werden und die man daher nicht einfach im Programmcode nachlesen kann.

2.3.6.1. Integer Literale

Die folgende Zeile enthält einen literalen Wert, den Wert 27, wie Sie leicht erkennen können.

```
int num32 = 27;
```

Wann immer der Compiler ein ganzzahliges Literal, wie hier 27, findet, und bevor irgend welche Operation mit dieser Zahl ausgeführt werden, wird eine *anonyme Konstante* vom Typ (Datentyp) `int` erzeugt, und der Wert (27) darin abgespeichert.

Als nächstes wird diese anonyme Konstante der Integer Variable `num32` zugewiesen. Das funktioniert ohne Probleme.

Es können aber Probleme auftauchen, die vom Java System abgefangen werden müssen. Betrachten wir dazu folgende Anweisung:

```
byte num8 = 27;
```

In diesem Fall sieht der interne Ablauf folgendermassen aus:

1. der Compiler kreiert eine Variable `num8` vom Datentyp `byte` (`num8` hat also maximal Vorzeichen plus 7 signifikante Bits).

2. der Compiler sieht das Integer Literal 27 und speichert es anonym im Integer Speicherbereich ab. Dazu kreiert der Compiler eine Variable vom Datentyp int (also vier Bytes oder 32Bits lang [1 Vorzeichenbit plus 31 Bits zum Speichern von Werten]).

3. jetzt folgt die Zuweisung dieser anonymen 32 Bit Integer Zahl an die 8 Bit lange num32 Variable: dabei findet eine Reduzierung der Bitlänge von 32 auf 8 Bits statt!
   Der Compiler prüft zuerst ob diese Operation legal ist:

   falls in den 3 Bytes, die abgeschnitten werden irgend wo eine Eins steht, dann wird die Operation *abgebrochen*, da in diesem Falle wichtige Informationen verloren gingen.

   in unserem Fall ist dies aber nicht so (d.h. diese Operation kann ausgeführt werden), da 27 in den letzten 8 Bits (dem letzten Byte) vollständig Platz hat.

   Schauen wir uns einmal die verschiedenen Stadien an:

   27 in interner Integer (=32 Bit) Bit-Darstellung :
   0000 0000 0000 0000 0000 0000 0001 1011

   27 danach als Byte (8 Bit):
   0001 1011

   Wie Sie sehen, geht dabei keinerlei Information verloren. Diese Operation ist also erfolgreich und legal.

2.3.6.2. Integer Literale - Herabstufung (Demotion)

Komplexer wird die Situation, wenn wir ein Integer Literal haben, welches grösser als der Zielbereich ist. Diesen Fehler erkennt das Java System und generiert eine Fehlermeldung.

Schauen wir uns diesen Fall einmal an:

```
byte num8 = 257;
```

Dies ist eine illegale Anweisung. Warum?
Schauen wir und wieder der Reihe nach an, was passiert:

1. der Compiler reserviert für die `byte` Variable `num8` Speicherplatz (1 Byte) im Speicherbereich der `byte` Variablen.

2. der Compiler sieht das Integer Literal 257 und generiert eine anonyme Integer Konstante mit dem Wert 257:
   0000 0000 0000 0000 0000 0001 0000 0001

3. der Compiler muss diese anonyme Integer Konstante der `byte` Variable `num8` zuweisen. Diese hat aber nur 8 Bits. Der Rest muss also vorne abgeschnitten werden:
   aus
   0000 0000 0000 0000 0000 0001 0000 0001
   wird
   0000 0001
   also etwa ganz anderes (nämlich 1 statt 257).

4. Der Compiler generiert einen Fehler

Diese Situation bezeichnet man als ***Demotion*** oder ***Herabstufung***. Der Programmcode wurde unter der Annahme geschrieben, dass der Compiler diese Reduktion der Variablenlänge von `int` auf `byte` gestatten würde. *Herabstufung ist unter keinen Umständen erlaubt.*

Ein ähnlicher Fehler kann auftreten, falls ein literaler Wert einer `short` Variable zugeteilt wird, falls der Wert ausserhalb des Wertebereichs von `short` liegt.

Sie haben auch die Möglichkeit den Compiler anzuweisen, dass eine Herabstufung durchgeführt werden soll. Aber wie Sie gerade gesehen haben, kann dies zu Problemen führen.
Java kennt den Begriff des Castings, der Typumwandlung. Beispiel:

```
int num32 = 257;
byte num8 = (byte)num32;
```

In diesem Fall wird explizit eine Typenumwandlung, ein Casting verlangt und durchgeführt. Der resultierende Wert von num8 wird, wie wir oben gesehen haben, aber 1 sein. Das ist wahrscheinlich nicht das, was der Programmierer erwartet hätte.

2.3.6.2.1. Casting - Typenumwandlung

Casting ist die Konvertierung von unterschiedlichen Datentypen. Dabei wird die Anzahl Bits, welche für die Speicherung der Werte zur Verfügung stehen, verändert.

Das Casting wird soo durchgeführt, dass der Namen des Zieldatentyps direkt vor der zu konvertierenden Variable in Klammern hingeschrieben wird.

Hier einige Beispiele:

```
byte  num8 = (byte)27;  // ohne Verluste
int   num32 = 27;
byte  numX = (byte)num32;
```

2.3.6.3. Integer Literale - Promotion

Promotion nimmt den Wert einer Variablen und macht ihn länger. Promotion benötigt kein Casting, da einfach Nullen vorne angesetzt werden.

Damit kann man irgendwelche Literale Werte in Variablen vom Typ `long` abspeichern. Beispiel:

```
long num64 = 26;
```

Sie können auch ein `L` (oder `l`, aber besser ist Grossbuchstaben wegen der Verwechslungsgefahr) an eine Zahl anhängen. Dadurch wird der Compiler gezwungen, diese Variable als unbenannte 64 Bit Konstante anzusehen.

Beispiel:

```
long num64 = 27L;
```

2.3.6.4. Fliesskomma Literale

Auch Fliesskommazahlen besitzen Literale. Diese Werte kann man auf unterschiedliche Art und Weise schreiben, inklusive der Engineering Notation mit dem grossen oder kleinen e.

Beispiele für gültige Fliesskomma (engl. *floating point*) Literalwerte:

```
27.9
279E5
27.9e6
```

Sie können auch ganzzahlige Literale an Fliesskomma Literale zuweisen. Dies führt auch zu Promotionen.

Beispiele:

```
float fp32 = 27;
double fp64a =27;
double fp64b = 27L;
```

Die ***Standardwerte*** für Literale sind:
Integer Literale werden in anonyme `int` Variablen abgespeichert
floating point Literale werden in anonyme `double` Variablen abgespeichert.

Das hat zur folge, dass beispielsweise folgende Anweisung falsch ist und zu einem Compilerfehler führt:

```
float fpoint32 = 27.9;
```

Dabei würde eine 32 Bit Variable (`float fpoint32`) gezwungen einen 64 Bit Wert (27.9 als double) aufzunehmen. Das kann schief gehen, also muss der Compiler reagieren.

Korrekt lässt sich diese Umwandlung durchführen durch Casting:

```
float fpoint32a = (float)27.9;      // float wird float zugewiesen oder
float fpoint32b = 27.9F;
```

2.3.6.5. Zeichen Literale

Ein Zeichenliteral besteht aus einem einzelnen Zeichen, in einfachen Anführungszeichen (').

Beispiele:

```
'a'
'Z'
'@'
```

Jedes Zeichen, welches Sie auf der Tastatur eingeben können, auch fremdländische, können in einer char Variable mit Hilfe eines Character Literals abgespeichert werden - aber wie sieht es aus mit Spezialzeichen wie Zeilenwechsel, Tabulatoren und ähnliche Steuerzeichen?

Java übernimmt die übliche Beschreibung der Steuerzeichen:

`'\t'`	ist	Tab
`'\r'`	ist	carriage return
`'\n'`	ist	newline

Sie können aber auch Unicode Zeichen einsetzen, die Sie gar nicht sehen können. Diese werden direkt als Unicode eingegeben, mit Hilfe von `'\u<hexadezimale Zahl>'`, beispielsweise `'\u1234'`.

2.3.6.6. Boolesche und Referenz Literale

2.3.6.6.1. boolean Literale

Boole'sche Literale können lediglich die Werte `true` und `false` annehmen. Diese beiden Werte sind zudem Schlüsselworte und müssen daher klein geschrieben werden.

Beispiel:

```
boolean
meineErsteBoolescheVariable;
meineErsteBoolescheVariable = true;
```

2.3.6.6.2. Referenzliterale

Sie können keine absolute Adresse für ein Objekt angeben. Daher können Sie schlecht von Referenz Literalen sprechen. Aber es gibt einen literale Referenzwert.

Oft muss man die Referenz auf ein Objekt auflösen. Dies geschieht, indem man diese Referenz auf `null`, die null Referenz (nichts) setzt.

```
Computer noObject = null; // kein computer mehr
```

2.3.6.7. String Literale

Strings, Zeichenketten, sind eigentlich keine Basisdatentypen (primitive Datentypen) sondern Objekte. Objekte besitzen in der Regel auch keine literalen Werte. Ausser Zeichenkettenobjekte oder Strings.

Zeichenketten Literale werden in doppelten Anführungszeichen angegeben (") .
Fall man lediglich ein Zeichen des Strings betrachtet spricht man auch von einem `short` String. Falls der String überhaupt keine Zeichen enthält, spricht man von einem leeren String oder einer `null` Referenz. Diese benötigt man recht oft, um beispielsweise auszudrücken, dass eine String Referenz (Variable) sich auf kein String Objekt bezieht (also keinen Wert [ausser `null`] hat).

Im folgenden Beispiel wird der Variable `animal`, einem `String` Objekt, der Literalwert `walrus` zugeordnet:

```
String animal = "walrus";
```

2.3.7. Variablen - Review

Die folgende Frage sollten Sie nach dem Durcharbeiten dieses Kapitels über Literale beantworten können:

```
Variablen werden mit dem Schlüsselwort new deklariert. Welches der folgende
Worte ergänzt den Satz?[3]


float
integer
boolean
reference
```

[3] reference

2.3.8. Hello World!

Wie in fast jeder anderen Programmiersprache kreiert man mit Java Applikationen. Die einfachste aller Applikationen, allgemein als Hello World Programm bekannt, schreibt einfach die Zeichenkette "Hello World" auf den Bildschirm.

Das Java Programm unten zeigt, wie dieser Programmcode aussieht.

```
1    //
2    // Sample HelloWorld application
3    //
4    public class HelloWorldApp {
5        public static void main (String args[]) {
6            System.out.println ("Hello World!");
7        }
8    }
```

Auf den nächsten Seiten beschreiben wir dieses winzige Programm.

2.3.8.1. Beschreibung von `HelloWorldApp`

Die ersten drei Zeilen sind Kommentare. die doppelten "forward slash" (//) zeigen an, dass der Rest der Zeile als Kommentar zu interpretieren ist.

Der besseren Lesbarkeit zu Liebe wurde vor dem Text und nach dem Text (Sample...) eine Leerzeile eingeführt.


```
1    //
2    // Sample HelloWorld application
3    //
4    public class HelloWorldApp {
5        public static void main (String args[]) {
6            System.out.println ("Hello World!");
7        }
8    }
```

Zeile 4 ist eine Anweisung, welche die Klasse `HelloWorldApp` deklariert

```
1    //
2    // Sample HelloWorld application
3    //
4    public class HelloWorldApp {
5        public static void main (String args[]) {
6            System.out.println ("Hello World!");
7        }
8    }
```

Zeile 5 ist eine Anweisung, welche die Methode main deklariert.

```
1    //
2    // Sample HelloWorld application
3    //
4    public class HelloWorldApp {
5        public static void main (String args[]) {
6            System.out.println ("Hello World!");
7        }
8    }
```

Diese Methode ist der Startpunkt des Programms (bei Applets wäre es `init`(), wie wir noch sehen werden).

`main` ist eine spezielle Methode. Der Java Interpreter muss genau eine `main()` Methode finden. Und diese muss genau so aussehen wie diese. Sonst kann das Programm nicht als Applikation ausgeführt werden.

Falls Sie dem Programm beim Starten Parameter übergeben (auf der Kommandozeile), werden diese in die Elemente des `String` Arrays *`args`* abgespeichert.

Da wir in diesem Beispiel keine Parameter angeben werden, wird `args` im Programm selbst nicht weiter verwendet. Aber in der Methodendeklaration darf `args` nie fehlen, sonst wird die Applikation nicht gestartet.

Zeile 5 enthält wichtige Schlüsselworte:

- `public`
  Dieses Schlüsselwort zeigt an, dass auf die Methode `main`() aus anderen Klassen heraus zugegriffen werden kann, die Methode ist `public`, öffentlich zugänglich.

- `static`
  Dieses Schlüsselwort zeigt dem Compiler an, dass die Methode `main`() zur Klasse gehört, also eine Klassenmethode ist und demnach keine Instanz der Klasse benötigt, um ausgeführt zu werden.

- `void`
  Dieses Schlüsselwort zeigt an, dass die Methode `main()` keinen Wert produziert, keine Rückgabe liefert. Diese Angabe ist sehr wichtig, da das Java System prüft, ab die korrekten Datentypen verwendet werden, insbesondere auch, ob die Rückgabewerte einer Methode mit dem Datentyp aus deren Deklaration übereinstimmt.

- `String args[ ]`
  Falls wir eine Applikation starten, an die wir Parameterwerte übergeben müssen, werden diese an den String Array `args[ ]` übergeben. Im Programm selber müssen wir unter Umständen diese Parameter (`String` = Zeichenketten) in den gewünschten Datentyp (`int, float,` ...) umwandeln.

Zeile 6 illustriert folgendes:

```
1    //
2    // Sample HelloWorld application
3    //
4    public class HelloWorldApp {
5        public static void main (String args[]) {
6            System.out.println ("Hello World!");
7        }
8    }
```

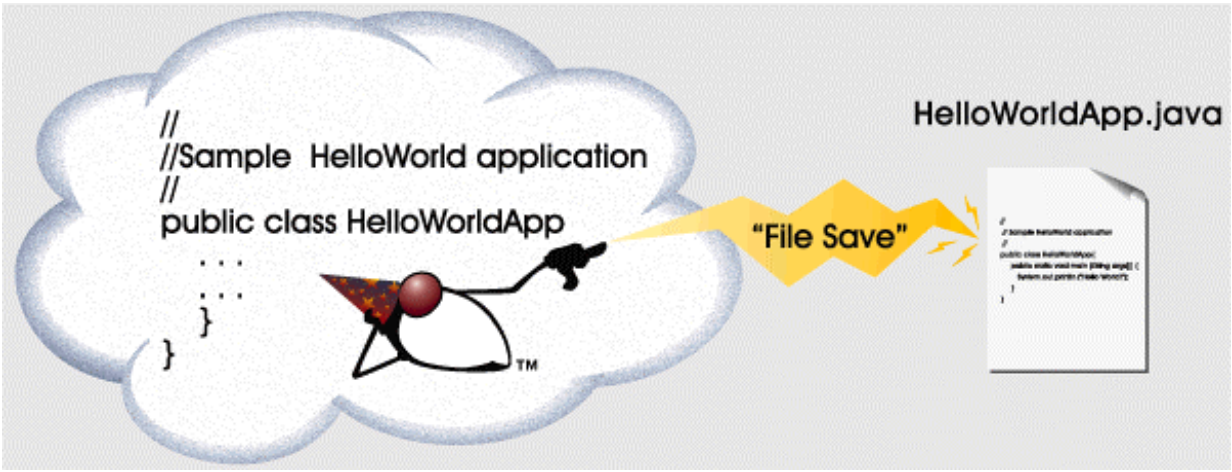
den Einsatz von

- einem Klassennamen (`System`)
- einem Objektnamen (`out`)
- einem Methodenaufruf (`println`)

Diese Zeile druckt die Zeichenkette "Hello World" aus, über die Standardausgabe, in unserem Fall auf den Bildschirm.
Zeile 6 und Zeile 7 schliessen die Methode `main` bzw. die Klasse `HelloWorldApp`.

2.3.8.2. Speichern der Applikation `HelloWorldApp`

Tippen Sie den Programmcode vom HelloWorld Programm im Wordpad oder einem Textverarbeitungssystem ein und speichern Sie das Programm in einer Datei mit dem Namen



`HelloWorldApp.java`

Generell müssen die Dateinamen mit den Klassennamen übereinstimmen.
Wenn Sie den Quellcode übersetzen, wird eine Datei

HelloWorldApp.class

generiert. Zum Übersetzen benutzen Sie den Java Compiler. Dieser wird im MS-DOS Fenster gestartet:

```
javac HelloWorldApp.java
```

Beachten Sie das "c" bei `javac`. `java` ohne "c" startet die virtuelle Maschine.

2.3.9. Entwickeln einer einfachen Java Applikation -Praxis

Nun sind Sie dran! Kreieren Sie eine Java Applikation, übersetzen Sie diese und starten Sie sie.
In dieser Übung müssen Sie eine einfache Applikation vervollständigen. Diese gibt einen einfachen Text aus.


```
//
// Ihr erstes Programm
//
public class MyProg {
      public static void main(String args[ ]) {
            // hier folgt Ihre Eingabe

      }
}
```

2.3.10. Übersetzen von `HelloWorldApp`

Wie Sie Ihre Applikation übersetzen und ausführen hängt von Ihrer Java Umgebung aus. Beispiele und Übungen dieser Kurseinheit illustrieren den Einsatz des Java Development Kits (JDK). Das JDK `javac` Kommando übersetzt Programmcode in Bytecode. Und der JDK Java Interpreter führt den Bytecode mit Hilfe des Befehls `java` aus.

Falls Sie den Programmcode für `HelloWorldApp.java` in diese Datei geschrieben haben, dann können Sie mit Hilfe des folgenden Befehls den Quellcode in `Bytecode` übersetzen:


```
javac HelloWorldApp.java
```

Falls der Compiler keine Meldung ausgibt, wird eine neue Datei mit dem Namen HelloWorldApp.class generiert und im selben Verzeichnis abgespeichert wie die Quelldatei.

2.3.10.1. Übersetzungsfehler

Es können sehr viele unterschiedliche Fehler während dem Übersetzungsvorgang auftreten. Die folgenden üblichen Fehler sind recht häufig:

- `javac: Command not found`
  Die Pfadvariable ist nicht oder nicht richtig gesetzt. Der Java Compiler wird nicht gefunden, mit andern Worten javac.exe wurde nicht gefunden. Dieses Programm befindet sich im bin Verzeichnis des JDK.

- `HelloWorldApp.java:3: Method printl(java.lang.String) not found in class java.io.PrintStream.`

```
                    System.out.printl ("Hello World!");
                                      ^
```

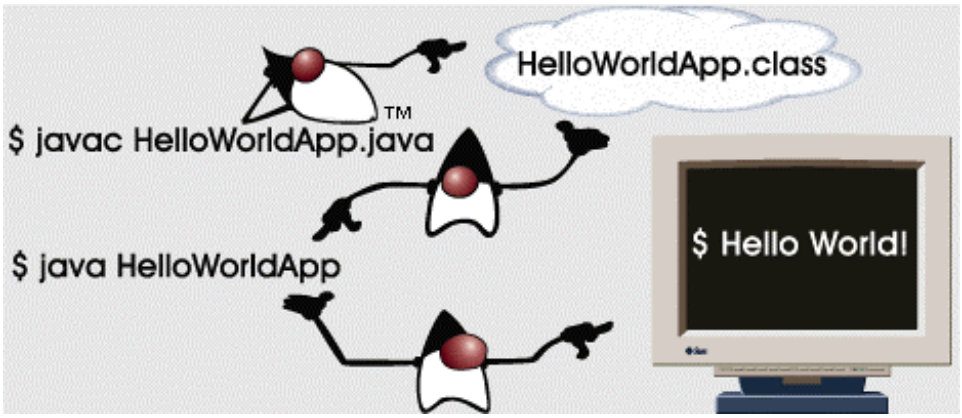
  Hier tritt ein Fehler auf, weil das 'n' in der Methode println() fehlt.

- `In class HelloWorldApp: main must be public and static`

  Dieser Fehler kann auftreten, falls entweder das Schlüsselwort `static` oder `public` fehlt.

2.3.11. Starten der `HelloWorldApp` Applikation

Um die HelloWorldApp Applikation zu starten, verwenden Sie den Java Interpreter. Deisen starten Sie mit dem Befehl `java`, gefolgt vom Programmname (=Name der .class Datei, wobei aber eben das Programm gestartet wird, also nicht der Dateiname angegeben wird. <programm_name>.class macht also überhaupt keinen Sinn).




```
$ java HelloWorldApp
Hello World!
```

Unter Umständen müssen Sie auch noch den Classpath, den Klassenpfad (= der Pfad unter dem die .class Dateien zu finden sind).

2.3.11.1. Laufzeit Fehler

Die folgenden Fehler können typischerweise während der Laufzeit auftreten:

- **`Can't find class HelloWorldApp`**

  Im Allgemeinen bedeutet dies, dass der Klassennamen falsch geschrieben wurde. Prüfen Sie, ob die Schreibweise stimmt.

- **`Naming`**

  Falls die Datei *.`java` eine public Klasse enthält, muss der Dateiname mit dem Klassennamen übereinstimmen.

- **`Class Count`**

  Pro Datei kann höchstens eine public Klasse definiert werden. Später werden wir sehen, unter welchen Umständen diese Regel durchbrochen werden kann.

2.3.12. Übersetzen und Starten einer einfachen Java Applikation - Praxis

Jetzt ist es an Ihnen praktisch zu üben.

Die folgenden Fragen sollten Sie aber nach dem Durcharbeiten dieses Moduls beantworten können.

1. Sie wollen die Applikation MeineApplikation, die Sie zuvor in der Datei MeineApplikation.java abgespeichert haben, übersetzen.
   Welchen Befehl geben Sie ein?[4]

   Ich nehme an, Sie haben die Applikation eventuell nach ein paar Korrekturen übersetzen können. In Ihrem Verzeichnis steht nun neben der Java auch noch die Class Datei mit dem übersetzten Bytecode.

2. Nun möchten wir die Applikation starten.
   Welchen Befehl geben Sie dafür ein?[5]

[4] javac MeineApplikation.java
[5] java MeineApplikation

2.3.13. Quiz - Einführung in Java

Das folgende Quiz besteht aus 6 Multiple-Choice oder wahr/falsch Fragen, mit denen Sie Ihr Verständnis einiger wichtiger Konzepte dieses Moduls testen können.

1. Welche der folgenden Anweisungen ist keine gültige Java Anweisung?[6]
   a) i = 6 + 9;
   b) { x=y-1, y=a+1;}
   c) middleInitial = 'L';
   d) int stars;

2. Sie können Kommentare nach folgenden Delimitern in Ihr Programm einfügen.[7]
   a) \\
   b) !!
   c) <!--
   d) a) und b) sind richtig
   e) keine ist richtig

3. Einige Schlüsselworte darf man als Namen von Klassen und Methoden verwenden.[8]
   a) trifft zu
   b) trifft nicht zu

4. In Java werden { } eingesetzt, um [9]
   a) Blöcke zu bilden
   b) Programmcode zu organisieren
   c) Ihren Programmcode lesbarer zu machen
   d) Methoden zu vereinbaren

5. Welcher der folgenden ist kein gültiger Java Identifier?[10]
   a) While
   b) 4_a_gift
   c) zähle_alle_Wörter_aus_diesem_Artikel
   d) $_Gehalt

6. Standard für ein Integer Literal ist [11]
   a) byte
   b) short
   c) int
   d) long

[6] b) da ein Semikolon fehlt (an Stelle des Kommas)
[7] e) korrekt wäre //
[8] b)
[9] a) Blöcke werden wie Anweisungen behandelt.
[10] b) weil sie mit einer Zahl beginnt
[11] int

2.3.14. Module Zusammenfassung

In diesem Modul haben wir einige Regeln und Werkzeuge kennen gelernt, welche Sie kennen müssen, bevor Sie mit der Java Programmiersprache arbeiten können.

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- die Regeln und Werkzeuge der Java Programmierumgebung einzusetzen
- Deklarationen und Anweisungen zu schreiben
- die Java Datentypen aufzuzählen
- die Literale der Basisdatentypen zu erkennen und zu verstehen, was in Java damit geschieht (einfache Umwandlungen Promotion, Demotion)
- übersetzen und ausführen eines Java Programms.

2.4. Module 2: Einfache Programmier Konstrukte

2.4.1. Einführung - Einfache Programmier Konstrukte

In diesem Modul werden Sie grundlegende Konstrukte kennen lernen, wie beispielsweise das `if` Konstrukt (man sagt dazu *Kontrollstruktur*, weil der Ablauf des Programms damit kontrolliert wird) und das `while` Konstrukt, mit dem Schleifen definiert werden können.

2.4.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- die `if` Kontrollanweisung in Programmen einzusetzen, um den Programmablauf aufgrund von Entscheidungen zu steuern.
- die `else` Kontrollanweisung in Programmen einzusetzen, um das Programm zu veranlassen einen alternativen Programmpfad zu durchlaufen.
- `if` und `else` Kontrollanweisungen zusammen anzuwenden, um die Ausführung eines bestimmten Programmblocks (oder im einfachsten Fall einer einzelnen Anweisung)
- mit Hilfe Boole'scher und arithmetischer Ausdrücke (mit Boole'schem Ergebnis) unterschiedliche Ausführungsbedingungen zu testen und den Programmablauf damit zu steuern.
- einfache mathematische Berechnungen im Programm auszuführen
- Inkrement- und Dekrement- Operation zu programmieren, um Werte von Variablen zu subtrahieren oder Werte zu Werten zu addieren.
- die `while` Kontrollstruktur in Programmen einzusetzen, um Schleifen zu programmieren.

2.4.1.2. Orientieren Sie sich selbst

Stellen Sie sich vor dem Durcharbeiten dieses Moduls folgende Fragen:

- Wie schreibe ich Programmcode, des es mir erlaubt, im Programm Entscheide zu treffen.
- Wie fälle ich selbst Entscheide? Wie sieht mein Entscheidungsprozess aus?

Überlegen Sie sich, ob Sie schon genug Kenntnisse haben, diesen Modul einfach zu überspringen.

Falls Sie den Modul durcharbeiten, sollten Sie die obigen zwei Fragen im Kopf behalten, sie helfen Ihnen beim Verständnis der folgenden Konstrukte.

2.4.2. Grundlegende Kontrollstrukturen

Wie kann der Programmablauf gesteuert werden?
Grundsätzlich brauchen wir die Möglichkeit, Teile des Programms abhängig von bestimmten Bedingungen auszuführen oder zu überspringen.
Daneben ist es für viele Anwendungen wichtig, bestimmte Teile eines Programms immer wieder oder öfters als einmal auszuführen.

2.4.2.1. Bedingungen - Konditionen

Die Möglichkeit, zur Laufzeit zu entscheiden, ob bestimmte Anweisungen ausgeführt werden sollen, ist sehr wesentlich für alle Programmieraufgaben. Wie die entsprechenden Konstrukte in Java aussehen, werden Sie im Folgenden kennenlernen.

2.4.2.2. Schleifen

Die Möglichkeit, zur Laufzeit zu entscheiden, wie oft eine bestimmte Anweisung (oder ein ganzer Block) ausgeführt werden soll., ist genaus so wichtig, wie Konditionen. Das grundlegende Java Konstrukt für diese Aufgabe ist `while`(Kondition). Java kennt *alternative* Konstrukte, Kontrollstrukturen, um Bedingungen abzufragen oder Schleifen zu programmieren. Falls Sie die Funktionsweise von `if` und `while` verstanden haben, können Sie auch die andern Kontrollstrukturen für Wiederholung und bedingte Ausführung anwenden.

Der Einfachheit halber werden wir im Folgenden in den Beispielen oft nur einfache Anweisungen betrachten; die Erweiterung auf ganze Anweisungsblöcke besteht darin, eine Anweisung durch einen Block ({... Anweisungen ...}).

2.4.3. Das `if` Statement

Das `if` Statement erlaubt es Ihrem Programm einfache Entscheide zu fällen, basierend auf der Auswertung von Boole'schen Ausdrücken. Ein Boole'scher Ausdruck ist ein Ausdruck, welcher genau zwei Werte annehmen kann und zwar `true` oder `false`.

```
if (boolean expression)
    statement;
```

Bei der Ausführung prüft die Java Virtual Machine (JVM), ob der Boole'sche Ausdruck wahr oder falsch ist. Falls der Ausdruck wahr ist, wird die Anweisung ausgeführt. Sonst wird die Anweisung ignoriert. Die Verallgemeinerung auf einen Block sieht folgendermassen aus:

```
if (boolean expression) {
 statement;
 statement;
 statement;
}
```

Oft ist es besser, wenn man der leichteren Lesbarkeit wegen, auch einzelne Anweisungen in Klammern setzt. Das sieht dann so aus:

```
if (boolean expression) {
    statement;
}
```

2.4.3.1. Wählen zwischen zwei Anweisungen

Oft muss ein Programm zwischen mehreren alternativen Ausführungspfaden auswählen. Falls ein Programm genau zwei Alternativen hat, lässt sich dies mit Hilfe des `else` Schlüsselwortes programmieren.

Zum Beispiel:

```
if (boolean expression) {
    statement;
} else {
    statement;
}
```

Diese Konstrukt kann man auch auf mehrere Anweisungen, also Anweisungsblöcke, anwenden.

```
if (boolean expression) {
    statement;
    statement;
    statement;
} else {
    statement;
    statement;
    statement;
}
```

Falls der Boole'sche Ausdruck den Wert `true` hat, wird die Anweisung, der Anweisungsblock, welcher der `if` Anweisung folgt, ausgeführt.

Falls der Boole'sche Ausdruck den Wert `false` hat, wird die Anweisung, der Anweisungsblock, welcher der `else` Anweisung folgt, ausgeführt.

2.4.3.2. Auswahl aus mehr als zwei Anweisungen

Sie können auch mehrere `if ... else` Konstrukte zusammensetzen, um komplexere Anweisungen zusammenzubauen.

Zum Beispiel:

```
if (boolean expression) {
    statement;
    statement;
} else if (boolean expression) {
    statement;
    statement;
} else if (boolean expression) {
    statement;
    statement;
}
```

```
if (boolean expression) {
    statement;
    statement;
} else if (boolean expression) {
    statement;
    statement;
} else if (boolean expression) {
    statement;
    statement;
} else {
    statement;
}
```

Das Konstrukt auf der linken Seite prüft jeden Boole'schen Ausdruck der Reihe nach, bis einer der Boole'schen Ausdrücke wahr ist; ist keiner der Ausdrücke wahr, wird keines der Anweisungen ausgeführt. Das Programm fährt nach den ganzen Abfragen einfach weiter.

Beim Konstrukt auf der rechten Seite, verfährt das Programm analog. Allerdings wird in diesem Fall, falls keine der Bedingungen wahr ist, die Anweisung nach `else` ausgeführt, da diese an keinerlei Bedingungen geknüpft ist.

2.4.3.3. Logische Operationen

Die folgende Tabelle fasst die unterschiedlichen Bedingungen zusammen, die in einem Programm testen können:

Operation	**Operator**	**Beispiel**
ist gleich wie	`==`	`if (i == 1)`
ist nicht gleich wie	`!=`	`if (i != 1)`
ist weniger als	`<`	`if (i < 1)`
ist weniger oder gleich wie	`<=`	`if (i <= 1)`
ist grösser als	`>`	`if (i > 1)`
grösser oder gleich wie	`>=`	`if (i >= 1)`

Um mehrere Boole'sche Ausdrücke zu kombinieren, benötigt man Verknüpfungsoperatoren für Boole'sche Werte:

Operation	**Operator**	**Beispiel**
AND	`&&`	`if ((i < 1) && (j > 6))`
OR	`||`	`if ((i < 1) || (j > 6))`
NOT	`!`	`if (!(i < 1))`

2.4.3.4. Auswertung Boole'scher Ausdrücke

Der Java Compiler wertet Boole'sche Ausdrücke gemäss folgender Tabelle aus:

Der **AND** Operator			logische UND Verknüpfung	
`&&`	**`true`**	**`false`**		
`true`	`true`	`false`	Resultat - true AND true = true	Resultat - true AND false = false
`false`	`false`	`false`	Resultat - false AND true = false	Resultat - false AND false = false

Der **OR** Operator			logische ODER Verknüpfung	
`||`	**`true`**	**`false`**		
`true`	`true`	`true`	Resultat - true OR true = true	Resultat - true OR false = true
`false`	`true`	`false`	Resultat - false OR true = true	Resultat - false OR false = false

Der **NOT** Operator		
`!`	**`true`**	**`false`**
	`false`	`true`

2.4.4. Das `if` Statement - Praxis

Diese Übung bietet Ihnen Gelegenheit, den Boole'schen Ausdruck einer Abfrage zu ergänzen:

Gegeben Sei folgender Programmcode. Ergänzen Sie ihn so, dass er zutrifft, falls die Temperatur über 90 Grad liegt.

```
int temp = 0;
...
if (<Ihr Ausdruck>)
```
[12]

Falls diese Bedingung zutrifft, wird eine passende Ausgabe gemacht, beispielsweise eine Textausgabe "Das ist aber sehr heiss".

Schreiben Sie den Code vollständig, ab der `if` Abfrage, inklusive Ausgabe.

Hier gleich die Musterlösung:

```
public class Temperatur {
      public static void main(String args[ ]) {
            int temp = 0;
            // Berechnung der Temperatur

            if (temp > 90) {
                  System.out.println("Das ist aber ganz schön heiss hier");
            }
      }
}
```

[12] temp>90

2.4.5. Arithmetische Konzepte

Damit Schleifen überhaupt sinnvoll eingesetzt werden können, benötigt man mathematische Operationen. Java bietet neben speziellen Paketen für mathematische Anwendungen fünf mathematische Grundoperationen (+ für Addition, - für Subtraktion, * für Multiplikation, / für Division und % für die Division mit Rest [Modulo Operation]).

Für die folgenden Beispiele seien

```
int num1 = 5;
int num2 = 3;
```

Das kann man in Java auch zusammenfassen zu

```
int num1=5, num2=3;
```

Operation	**Operator**	**Beispiel**	**Resultat**
Addition	+	sum = num1 + num2	8
Subtraktion	-	diff = num1 - num2	2
Multiplikation	*	prod = num1 * num2	15
Division	/	quot = num1 / num2	1 &Rest
Modulo (Rest)	%	mod = num1 % num2	2 &Vielfaches

Operator Präzedenz - Alle komplexen arithmetischen Berechnungen werden in folgender Reihenfolge ausgewertet:

1. Operatoren innerhalb von Klammern.
2. Multiplikations- und Divisions- Operatoren.
3. Additions- und Subtraktions- Operatoren.
4. falls der selbe Operator mehrfach vorkommt, wird der Ausdruck von links nach rechte ausgewertet.

2.4.5.1. Warum man Operatorpräzedenz benötigt

Das folgende Beispiel zeigt, warum Operatorpräzedenz nötig ist:

```
c = 25 - 5 * 4 / 2 - 10 + 4;
```

Ohne Operatorpräferenz wäre die Auswertung dieses Ausdrucks nicht auf eindeutige Art und Weise möglich. Mit den obigen Regeln liefert der Ausdruck das Ergebnis c = 9.
Mit Hilfe von Klammern lässt sich die Struktur leicht besser darstellen:

```
c = 25 - (5 * (4 / 2)) - 10 + 4;
```

Viele Programmierer setzen Klammern, um die Lesbarkeit des Programms zu erhöhen:

```
c = (((25 - 5) * 4) / (2 - 10)) + 4;
c = ((20 * 4) / (2 - 10)) + 4;
c = (80 / (2 - 10)) + 4;
c = (80 / -8) + 4;
c = -10 + 4;
c = -6;
```

2.4.6. Inkrement und Dekrement Operatoren

Die Addition und Subtraktion sind wohl die am häufigsten benötigten Operationen in einem Programm:


```
count = count - 9;
```

```
testScore = testScore + 5;
```

Besonders oft benötigt man die Erhöhung oder die Reduktion einer Variable um eins (1). Da dies sehr oft vorkommt, wurden spezielle Operatoren eingeführt, die genau dies tun:

Aufgabe	**Operator**	**Beispiel**	**falls i=5**
Pre-Inkrement	`++`	`j = ++i;`	`zuerst wird i um eins erhöht und dann j zugewiesen: i=6; j=6;`
Post-Inkrement	`++`	`j = i++;`	`zuerst wird i j zugewiesen und dann um eins erhöht: j=5; i=6;`
Pre-Dekrement	`--`	`j = --i;`	`zuerst wird i um eins vermindert und dann j zugewiesen: i=4; j=4;`
Post-Dekrement	`--`	`j = i--;`	`zuerst wird i j zugewiesen und dann um eins reduziert: j=5; i=4;`

Inkrement und Dekrement Operatoren kann man *auch innerhalb anderer Ausdrücke* anwenden. Zum Beispiel:

```
int i = 6;
int j = ++i;
```

In der ersten Zeile wird die Integer Variable i definiert und initialisiert (es wird ihr der Wert 6 zugewiesen).

In der zweiten Zeile wird eine Integer Variable j deklariert und ihr der Wert ++i zugewiesen.

Da hier der präinkrement Operator verwendet wird, heisst dies, dass zuerst die Variable i um eines erhöht du anschliessend dieser neue Wert der Variablen j zugewiesen wird. i wird als 7 und dieser Wert wird dann der Variable j zugewiesen.

Wenn wir an Stelle des prä- den post-inkrement Operator einsetzen, also

```
int i = 6;
int j = i++;
```

Der Wert der Variablen sieht allerdings nun anders aus: zuerst wird i der Wert 6 zugewiesen. Dann wird der Variable j auch der Wert zugewiesen und schliesslich i um eins auf 7 erhöht.

2.4.7. Der Inkrement Operator - Praxis

Nun liegt es an Ihnen zu zeigen, ob Sie das Wesentliche über inkrement und dekrement Operatoren verstanden haben.

Die folgende Übung dient einzig und allein der Wissensprüfung.

1. Geben sei die Deklaration und Initialisierung der Variable i mit dem Wert 7. Deklarieren Sie eine Variable j, welche um eins grösser ist als i.

   int i=7;
   j= ?;[13]

2. Musterlösung:
   Nach der Ausführung von ++i wird i zu 8 und dieser Wert wird j zugewiesen.

[13] int j=++i;

2.4.8. Die `while` Schlaufe

Schleifen sind sinnvoll, falls man Programmblöcke mehrfach ausführen muss. Die `while` Schleife kann dafür eingesetzt werden.

Bei der `while` Schleife wird eine Bedingung getestet. Falls der Test vor der Schleifenausführung den Wert `true` liefert, wird der Block nach dem `while` ausgeführt. Danach wird der Test erneut ausgeführt. Dieser Prozess wird solange ausgeführt, bis die Bedingung nicht mehr erfüllt ist.

Danach wird mit der Ausführung direkt nach dem Block weitergefahren. Falls dies der Fall ist, wird also keine der Anweisungen im Block zur Schleife mehr ausgeführt.

Da im Extremfall die Schleife nie ausgeführt wird, spricht man auch von einer "zero or many" iterativen Schleife.

```
while (condition) {
    statement;
    statement;
}
```

Wenn wir beispielsweise einen Block mehrere Mal ausführen möchten und eine Variable die Werte 10, 20, 30 ...90 annehmen sollte, könnte der Programmrumpf für diesen Teil etwa folgendermassen aussehen:

```
int count = 10;
while (count <= 100) {
    // Anweisung(en)
    count = count + 10;
}
```

Im ersten Durchlauf wird die Variable `count =10`; sie wird aber in diesem Durchgang auf 20 erhöht (`count = count + 10;`);
die Prüfung der Bedingung bei der `while` Schleife ergibt also `20 <=100` also true.
Damit ist die Bedingung für die `while` Schlaufe erfüllt und der Block wird erneut ausgeführt.

Sobald die Variable bei 100 angelangt ist, wird die Schleife das letzte Mal ausgeführt und anschliessen mit den Anweisungen nach dem `while` Block, also der (hier nicht gezeigten) Anweisung nach der schliessenden Klammer `}` weitergefahren.

2.4.9. Quiz

Die folgenden Testaufgaben sollten Sie nach dem Durcharbeiten dieses Moduls lösen können.

Der Test umfasst die bedingte Anweisung und die Schlaufe.

Ihre Aufgabe ist es, entweder den korrekten Code auszuwählen oder zu ergänzen.

1. Welches der folgenden Konstrukte ist eine korrekte `if` Anweisung?[14]

   a)
   ```
   if (count == 10) ; {
         count = count + 1;
     }
   ```

   b)
   ```
   if (count = 4) {
         count = count -2;
     }
   ```

   c)
   ```
   if (count == 7) {
         count = count + 1;
      }
   ```

2. Eine `while` Schlaufe [15]

   a) führt eine Anweisung / einen Anweisungsblock aus, erhöht einen Zäher und führt die Schleife erneut aus, bis der Zähler einen bestimmten Wert erreicht hat.

   b) fährt so lange fort, wie die Schleifenbedingung `false` ist.

   c) prüft eine Bedingung und führt, falls diese Bedingung zutrifft, die Schleife solange aus, wie die Bedingung zutrifft.

3. Welcher Wert besitzt `num` nach Ausführung der folgenden Anweisungen?[16]
   ```
   int num, num1;
   num1 = 5;
   num = num1++;
   ```
   a) 5
   b) 6
   c) 7

[14] c) : bei a) ist nach der Abfrage ein Semikolon zu viel; bei b) fehlt ein = Zeichen (es wird keine Zuweisung gemacht, sondern verglichen).
[15] c)
[16] a) : num1 wird erst nach der Zuweisung erhöht.

4. In Java können Sie beliebig viele `else` zu einem `if` kombinieren, also beliebig viele Alternativen programmieren.[17]

   a) wahr
   b) falsch

5. Welche der folgenden Operatoren bedeutet "ungleich"?[18]
   a) >=
   b) <=
   c) <>
   d) !=

6. Welches der folgenden Symbole beschreibt den logischen ODER Operator?[19]
   a) !!
   b) &&
   c) ||
   d) ¦¦

[17] b) : if...else...else.... else... ist nicht gestattet; if - else if ... else ... ist gestattet
[18] d)
[19] || oder c)

2.4.10. Modul Zusammenfassung

In diesem Modul haben Sie die `if` Bedingtanweisung und die `while` Schleifenanweisung, zwei Kontrollstrukturen, sowie verschiedene mathematische Operatoren kennen gelernt.

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein,

- das `if` Konstrukt anzuwenden, um innerhalb des Programms Entscheide zu fällen
- das `else` Schlüsselwort einzusetzen, um alternative Ausführungspfade in Ihren Programmen anzugeben.
- `if` und `else` Strukturen zu kombinieren, um ganze Serien von Abfragen miteinander zu verknüpfen.
- mit Hilfe von Boole'scher Logik und arithmetischen Operatoren Schleifen und Abfragen zu steuern.
- Inkrement und Dekrement Operatoren korrekt anzuwenden, um den Wert einer Variable um eins zu erhöhen oder zu reduzieren.
- das `while` Konstrukt einzusetzen, um Teile Ihres Programms mehrfach auszuführen.

2.5. Module 3: Fortgeschrittene Java Sprachkonzepte

In diesem Modul wollen wir weitere Sprachkonzepte, neben if und while, die wir bereits kennen, kennenlernen.

Dabei geht es um komplexere Abfragen und Schleifenkonstrukte.

2.5.1. Einführung

Die while Schleife und die if stellen Kontrollanweisungen und Auswahlkonstrukte dar, um beliebige Programmieraufgaben lösen zu können. Aber es gibt noch zusätzliche Konstrukte, mit deren Hilfe Programme einfacher geschrieben werden können, und die Programme auch lesbarer machen.

Die folgenden Kontrollanweisungen werden in diesem Modul vorgestellt:

- `for` Schleifem - zur Kontrolle, wie oft Schleifen durchlaufen werden
- `do` Schleifen - mit deren Hilfe eine Situation beschrieben wird, bei der eine Schleife mindestens einmal durchlaufen wird.
- `switch` Anweisung - mit deren Hilfe Verzweigungen effizienter programmiert werden können und komplexe `if` Verschachtelungen vermieden werden.
- `break` Anweisung - mit deren Hilfe der Ablauf (der Kontrollfluss) innerhalb des Programms in Schleifen kontrolliert werden kann. break führt zum Abbruch einer Schleife, das Programm wird am Ende der Schleife fortgesetzt.
- `continue` Anweisung - mit deren Hilfe der Ablauf innerhalb des Programms in Schleifen kontrolliert werden kann. `continue` unterbricht die aktuelle Schleife und fährt mit der nächsten Schleife fort.

2.5.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein,

- die `for` und `do` Kontrollanweisungen einzusetzen, um Teile Ihres Programms mehrfach durchlaufen zu lassen
- die `switch` Kontrollanweisung einzusetzen, um Entscheide in Ihrem Programm einzubauen.
- die `break` Kontrollanweisung einzusetzen, um Schleifen und `switch` Anweisungen zu beenden.
- die `continue` Kontrollanweisung einzusetzen, um eine Iteration an einer bestimmten Stelle im Programm abzubrechen (und mit der nächsten Schleife zu beginnen).

2.5.1.2. Orientieren Sie sich selbst

Bereiten Sie sich auf diesen Modul vor, indem Sie versuchen folgende Fragen zu beantworten:

- wie schwierig ist es komplexe `if - else` Abfragen zu programmieren und zu testen?
- wie können Sie eine Schleife verlassen?
- wie programmiere ich eine Schleife, falls ich genau weiss, wie viele Iterationen ich benötige?

Fragen Sie sich, was Sie zu diesem Thema bereits wissen.

Falls Sie überzeugt sind, bereits genügend Kenntnisse zu diesem Thema zu haben, können Sie den Modul auch überspringen.

Falls Sie sich entschliessen, den Modul durchzuarbeiten, dann sollten Sie diese Fragen im Kopf behalten. Sie helfen Ihnen beim Durcharbeiten.

2.5.2. Die `for` Schleife

Das Funktionsprinzip der while Schleife bestand aus drei Teilen:
1. setzen einer Bedingung
2. prüfen ob die Bedingung eintrifft und falls ja
3. einen Ausführungsblock ({...}) *solange* ausführen wie die Bedingung wahr ist.

Die for Schleife gestattet es dem Programmierer, einen Ausführungsblock eine *bestimmte Anzahl* auszuführen. Sonst besteht kein Unterschied zur while Schleife, ausser dass die Schleife mit Hilfe eines Zählers eine bestimmte Anzahl mal ausgeführt wurde.

Syntax der `for` Schleife:

```
for (initialisieren; testen eines Ausdrucks; Zähler erhöhen) {
    Rumpf;
}
```

Initialisieren
Diese Anweisung wird nur einmal verarbeitet und zwar bevor irgend eine andere Anweisung ausgeführt wird. Der Schleifenzähler wird lediglich auf den Startwert gesetzt (und gegebenenfalls deklariert als lokale Variable). Mehrere Initialisierungen werden, falls vorhanden, durch ein Komma getrennt.

Testen eines Ausdrucks
Mit dieser Anweisung wird getestet, ob eine weitere Schleife durchlaufen werden soll. Die Anweisung muss also einen Boole'schen Wert zurück liefern. Falls der Ausdruck `false` ist, wird die Schleife nicht mehr ausgeführt und mit der Anweisung ausserhalb des Rumpfes fortgefahren.

Zähler erhöhen
Diese Anweisung wird ausgeführt nachdem der Rumpf ausgeführt wurde, aber bevor getestet wird, ob ein weiterer Durchlauf benötigt wird. Falls man mehrere Zähler hat, müssen die einzelnen Zähler jeweils durch ein Komma getrennt sein.

Rumpf
Dieser Teil des Programms wird immer wieder ausgeführt, solange die Schleife durchlaufen wird.

Beispiel:

```
for (count = 0; count <= 2; count ++ ){
    System.out.println("count="+count);
}
```

Ausgabe:

count = 0
count = 1
count = 2

2.5.3. Die `do` Schleife

Eine `while` Schleife gestattet das Setzen von Bedingungen, die erfüllt sein müssen, damit überhaupt etwas geschieht, ausgeführt wird. Die Bedingung wird *vor dem Ausführen* der Anweisung(en) getestet. In einer `do` Schleife wird die *Bedingung nach der Ausführung* getestet, die Anweisungen werden also mindestens einmal ausgeführt.

Das folgende Beispiel zeigt eine do Schleife, welche einen Anweisungsblock einmal ausführt und dann wiederholt, bis x gleich 10 ist:

```
int x = 1;
do {
    //body statements
        x++;
} while (x < 10);
```

Bemerkung - Das Semikolon nach der Bedingung muss vorhanden sein. `for` und `while` Anweisungen verarbeiten eine einzelne Anweisung oder einen Block und enden damit entweder mit einem Semikolon oder mit einer schliessenden Klammer.

2.5.3.1. Die `do` Schleife versus die `while` Schleife

Der Einfachheit halber betrachten wir ein Beispiel und erklären den Unterschied an diesem Beispiel.

Wir betrachten ein einfaches Würfelspiel mit zwei Würfeln. der zweite Würfel muss solange gewürfelt werden, bis er dieselbe Zahl wie der erste Würfel zeigt.

Die folgenden Konstrukte zeigen die Lösung mit je einer der Schleifen- Anweisungen:

```
1  int würfel1, würfel2;
2  würfel1 = würfle(würfel1);
3  würfel2 = würfle(Würfel2);
4  while (würfel1 != würfel2) {
5   würfel2 = würfle(Würfel2);
6 }
```

Zeile 3 und 5 sind gleich: der Würfel2 muss mindestens einmal gewürfelt werden. Das gleiche Programm lässt sich auch mit einer `do` Schleife beschreiben:

```
1  int würfel1, würfel2;
2  würfel1 = würfle(würfel1);
3  do {
4  würfel2 = würfle(würfel2);
5 } while (würfel1 != würfel2);
```

Hier erscheint die Anweisung in Zeile 4 lediglich einmal, weil mit der `do` Schleife sicher gestellt wird, dass die Schleife mindestens einmal durchlaufen wird.

2.5.4. Die `do` Schlaufe - Praxis

In dieser Übung haben Sie die Möglichkeit, die do Schleife einzusetzen.

Ihre Aufgabe ist es, das vorgegebene programm zu ergänzen.

Die Musterlösung finden Sie, wie immer, in der Fussnote.

Viel Erfolg!

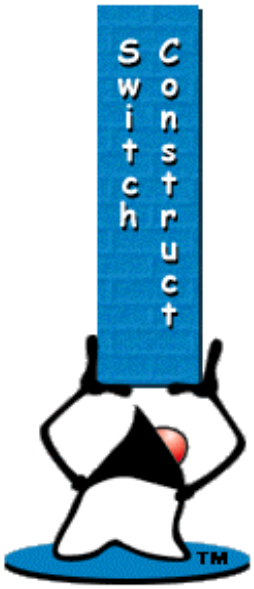
1. Ergänzen Sie den folgenden Programmcode durch eine `while` Anweisung, so dass die Schleife ausgeführt wir, bis x nach Abschluss der Schleife den Wert `10` hat.

```
x = 2;
do {
   x++;
} while (...);
```
[20]

[20] while(x<10);

2.5.5. Das `switch` Konstrukt

Schauen Sie sich das folgende Konstrukt an. Es überprüft den Inhalt einer Variable `i`:




```
int i = someValue();
if (i == 1) {
    statementA();
} else if (i == 2) {
    statementB();
    statementC();
} else if ((i == 3) || (i == 4)) {
    statementC();
} else if (i == 5) {
    statementD();
    statementE();
} else {
    statementF();
}
```

Die vielen `if` Abfragen machen die Abfrage schwer lesbar und unübersichtlich. Das obige `if` Konstrukt ist zu kompliziert. Es führt auch zu redundantem oder wiederverwendeten Programmcode (`statementC()` und mehrere `if..else`)

Einfacher lässt sich das Programm mit Hilfe von `switch` schreiben, vorallem übersichtlicher.

2.5.5.1. Einsatz von `switch` an Stelle von `if`

```
int i = someValue();
if (i == 1) {
    statementA();
} else if (i == 2) {
    statementB();
    statementC();
} else if ((i == 3) || (i == 4)) {
    statementC();
} else if (i == 5) {
    statementD();
    statementE();
} else {
    statementF();
}
```

```
1  int i = someValue();
2  switch (i) {
3  case 1:
4      statementA();
5      break;
6  case 2:
7      statementB();
8  case 3:
9  case 4:
10     statementC();
11     break;
12 case 5:
13     statementD();
14     statementE();
15     break;
16 default:
17     statementF();
18 }
```

Zeile 1:	Setze `i` auf `someValue`.
Zeile 2:	Bedeutet "vergleiche alle Fälle mit dem Wert von `i`." (Beachten Sie, dass `case` das selbe bedeutet wie im Englischen.)
Zeile 3 & 4:	Bedeutet "falls (in case) der Wert von `i` gleich 1 ist, führe `statementA( )` aus und `break`."
Zeile 5:	Das Schlüsselwort `break` bedeutet, "break out aus der Struktur und fahre mit dem Programm nach Zeile 18 weiter." Beachten Sie: falls `someValue` nicht gleich 1 ist, werden die Zeilen 4 und 5 nicht ausgeführt.) Falls man die `break` Anweisungen weglassen würde, dann würden, falls `i = 1`, alle Anweisungen ausgeführt.
Zeile 6 - 11:	Im `if` Beispiel (links), beachten Sie, dass falls `i` gleich 2 ist, dann beide Anweisungen `statementB( )` und `statementC( )` ausgeführt werden. Dies wird dadurch erreicht, dass rechts die `break` Anweisung in `case 2` fehlt, wodurch `statementB` und `statementC` ausgeführt werden, bevor eine `break` Anweisung in Zeile 11 angetroffen wird. Falls `i` nicht 2 ist, dann soll geprüft werden, ob `i` 3 oder 4 ist. In diesem Fall wird nur die Anweisung `statementC( )` ausgeführt. (Beachten Sie auf Zeile 8 dass eine Anweisung ohne Einschränkung wie ein `or` Operator ('||') für zwei Literale agiert.)
Zeile 16:	Falls keiner der Fälle eintrifft, soll dieser Block ausgeführt werden. Dieser Teil der `switch` Anweisung entspricht dem `else` bei der `if` Anweisung. Dieser Teil kann auch fehlen. Aber falls `i` nicht im Bereich von 1 bis 5 ist, "fällt das Programm durch" die `switch` Anweisung, ohne dass irgend etwas geschah. Das ist aber vielleicht nicht die Absicht des Programmierers.

2.5.5.2. Die `switch` Anweisung - Regeln & Fakten

- die Variable i kann vom Typus `char`, `byte`, `short`, oder `int` sein; das heisst, die Werte sind `case 1`, `2`, `3`, etc., oder `case a`, `b`, `c`, etc.
- das `case` Label muss ein Literal sein; Variablen, Ausdrücke oder Methodenaufrufe sind nicht gestattet.
- das `case` Label ist ein Entry Point für eine ganze Sequenz von Anweisungen.

2.5.6. Die `break` Anweisung

Die `break` Anweisung gestattet den kontrollierten und unmittelbaren Abbruch einer Schleife oder einer `switch` Anweisung. Bei der `switch` Anweisung wird `break` eingesetzt, um das "zwischen durchfallen" zu vermeiden, also die Ausführung mehrerer hintereinander stehender `case` Blöcke.

Die folgende `for` Schleife versucht von 1 bis 50 zu zählen. Falls die Zahl durch 15 teilbar ist, soll ein Zähler erhöht werden; falls dieser Zähler 3 ist, soll die Schleife unterbrochen werden.

Die Teilbarkeit wird mit Hilfe des Modulo Operators (%) berechnet. Falls diese Modulo Division den Wert 0 liefert, also die Zahl teilbar ist, wird der Zähler um eines erhöht.

Die `break` Anweisung ist nur innerhalb `while`, `for`, `do`, und `switch` Konstrukten gestattet.

```
int mult15Count = 0;
for (int i = 1 ; i < = 50 ; i++ {
    if ((i % 15) == 0) {
        mult15Count++;
    if (mult15Count == 3) {
        break;
    }
    }
}
// hier wird nach dem break weitergefahren
```

2.5.7. Die `continue` Anweisung

Die `continue` Anweisung gestattet einen kontrollierten unmittelbaren Abbruch einer Iteration einer Schleife. Aber im Gegensatz zur `break` Anweisung, die aus der Schleife springt, gestattet die `continue` Anweisung den Abbruch der aktuellen Schleife. Es wird also mit der nächsten Schleife fortgefahren (sofern die Bedingung dazu erfüllt ist).

Im folgenden Beispiel eines `for` Konstrukts wird von 1 bis 50 gezählt, wobei Mehrfache von 9 gezählt werden.

Falls die Zahl durch 9 teilbar ist, wird der Zähler `mul9Count` um eins erhöht.
Falls die Zahl nicht durch 9 teilbar ist, wird die Schleife abgebrochen und mit dem nächsten Schleifendurchgang weitergefahren.

```
int mult9Count = 0;
for (int i = 1; i <= 50 ; i++) {
    if ((i % 9) ! = 0) {
        continue;  /* spring an das Ende der Schleife
                      Damit wird i um eins erhöht
                      und die Schleife fortgesetzt
                   */
    }
    mult9Count = mult9Count + 1;
}
```

Die `continue` Anweisung ist innerhalb von `while`, `for` und `do` Schleifen gestattet.

2.5.8. Die `switch` Anweisung - Praxis

Jetzt können Sie prüfen, ob Sie diesen Modul so durchgearbeitet haben, dass Sie auch konkret das Wissen einsetzen können.

Die Übung ist im üblichen Stil:
Sie müssen ein Programm ergänzen.

1. Gegeben sei folgender Programmcode:

```
int x = 3;
switch(x) {
...
...
...
...
```

   Ergänzen Sie das Programm so, dass ein Block ausgeführt wird, falls die Variable `x`, wie oben den Wert 3 hat.[21]

2. Jetzt ergänzen Sie den obigen Programmcode plus Musterlösung so, dass nach der Ausführung der Anweisung die switch Anweisung verlassen wird:[22]

```
int x = 3;
switch(x) {
   case 3 :
         Anweisung3;
         ...
...
...
...
```

3. Damit sieht die Musterlösung folgendermassen aus:

```
int x = 3;

switch(x) {
   case 3:     Anweisung3;
               break;
   default:    DefaultAnweisung;
               break;
}
```

[21] case 3:
[22] break;

2.5.9. Quiz - Fortgeschrittene Java Sprachkonzepte

Jetzt liegt es an Ihnen zu beweisen, dass Sie den Inhalt dieses Moduls verstanden haben.

Sie können nicht alle Konstrukte, die wir besprochen haben, einüben. Aber falls Sie wirklich Java programmieren lernen, sollten Sie doch einige zusätzliche Programme schreiben und die Konstrukte üben.

1. Im folgenden Programm wird die `for` Schleife korrekt wiedergegeben:[23]

```
for (initialize, test expression, increment counter) {
   body;
}
```

   a) wahr
   b) falsch

2. Falls Sie einen Block mindestens einmal ausgeführt haben möchten, welches Schleifenkonstrukt ist dann am Besten angepasst:[24]

   a) while Schleife
   b) for Schleife
   c) do Schleife

3. Mit welcher Anweisung kann eine Schleife unterbrochen werden, ohne dass die gesamte Schleifenkonstruktion abgebrochen und verlassen wird?[25]

   a) `continue`
   b) `break`

4. Wann können Sie die `continue` Anweisung nicht einsetzen?[26]

   a) `do`
   b) `while`
   c) `for`
   d) `switch`

[23] b) falsch
[24] c) : bei diesem Konstrukt wird die Schleife mindestens einmal ausgeführt
[25] a) : mit continue wird nur die aktuelle Schleife abgebrochen
[26] d) : continue macht beim switch Statement keinen Sinn (es ist in allen andern Fällen einsetzbar)

2.5.10. Module Zusammenfassung

In diesem Modul sind wir über die einfache `if` Anweisung und `while` Schlaufen hinausgegangen. Dieser Modul präsentierte komplexere bedingte (konditionelle) und Schleifen- Anweisungen mit denen Sie komplexere Programmlogiken realisieren können.

Nach dem Durcharbeiten sollten Sie in der Lage sein:

- die `for` und `do` Einzusetzen, um Schleifen zu programmieren
- die `switch` Anweisung einzusetzen, um in Ihren Programmen Entscheide besser strukturieren zu können.
- die `break` Anweisung einzusetzen, um Schleifen und `switch` Anweisungen kontrolliert abzubrechen.
- die `continue` Anweisung einzusetzen, um eine Iteration einer Schleife zu beenden.

2.6. Module 4: Kapselung

Kapselung (engl. *encapsulation*) ist eines der Grundkonzepte des objektorientierten Entwurfs. Dieses Konzept sorgt ganz wesentlich dafür, dass Objekte unabhängig werden und damit viel besser mehrfach eingesetzt werden können.

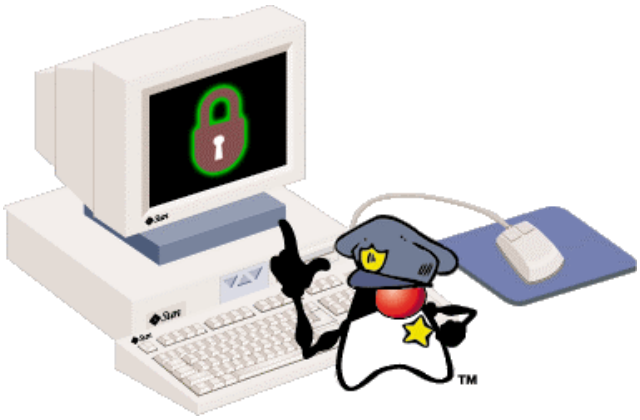
2.6.1. Einleitung

Kapselung dient neben der bewussten starken Modularisierung dazu bei, dass Daten in den Objekten vor externen Zugriffen geschützt werden können.

2.6.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein,

- den Zugriff auf die Datenfelder der Objekte mit Hilfe der Kapselung einzuschränken
- mit Hilfe der Schlüsselworte public und private den Zugriff auf Datenfelder zu steuern.



2.6.1.2. Orientieren Sie sich selbst

Wie immer sollten Sie vor dem Durcharbeiten Ihre Gedanken in die richtige Richtung lenken und sich einige Fragen durch den Kopf gehen lassen.

- wie würden Sie den Zugriff auf sensitive Daten Ihrer Objekte schützen?
- wie würden Sie auf der andern Seite den Zugriff auf die Daten, die Sie extern benötigen, zulassen?
- wie würden Sie den Zugriff auf Methoden kontrollieren?

2.6.2. Zugriffseinschränkung durch Kapselung

Kapselung ist das Prinzip, bestimmte Members (Data Members oder Attribute oder Datenfelder; Function Members oder Methoden) Ihrer Objekte vor unberechtigten externen Zugriffen zu schützen. Objekte sind selbständige Einheiten generischer Klassendefinitionen und werden oft Daten enthalten, welche als privat und damit geschützt oder schützenswert angesehen werden müssen.

Als Beispiel können Sie sich die Beschreibung eines Angestellten einer Firma vorstellen. Dieses Objekt enthält die Personalien des Angestellten:

```
class Employee {
    int employeeNumber;
    String name;
    int departmentNumber;
    int extensionNumber;
    int salary;
    // und so weiter
}
```

Die Zugriffsrechte auf dies Datenfelder, die Data Members, dieser Objekte ist recht unterschiedlich. Den Namen können wesentlich mehr Personen abfragen als beispielsweise das Gehalt. Auch die Mutation der Daten kann in der Regel lediglich von einem bestimmten Personenkreis vorgenommen werden.

Das Gehaltsfeld wird in der Regel sogar in irgend einer Form verschlüsselt werden müssen. Falls die Zugriffsrechte nicht strikt festgelegt werden, muss die Firma oder Organisation mit einer Klage wegen Verletzung der Privatsphäre und des Datenschutzes rechnen.

Java stellt einige Zugriffs-'Modifier' zur Verfügung, mit deren Hilfe bestimmte, aber noch recht grobe Zugriffsrechte vergeben werden können. Später werden wir uns genauer mit Sicherheitsaspekten von Java befassen, unter anderem mit der sogenannten Java Security Policy.

2.6.2.1. Das `public` Schlüsselwort

Java stellt einige 'Modifier' zur Verfügung, mit deren Hilfe der Zugriff auf den Inhalt der Objekte eingeschränkt werden kann. Sie haben beispielsweise die Möglichkeit, allen den Zugriff auf die Daten und die Methoden eines Objekts zu gestatten; oder aber den Zugriff zu verbieten. Dann kann nur noch das Objekt selbst auf seine Daten zugreifen.

Das Schlüsselwort `public` 'öffnet' die Objekte; das Schlüsselwort `private` 'schliesst' sie. Sowohl Methoden als auch Datenfelder können diese Attribute tragen, vorangestellt.

Schauen wir uns ein Beispiel an:

```
class PublicExample {
    public static void main(String args[ ]) {
        Employee tempEmployee = new Employee( );

        tempEmployee.employeeNumber = 27;
        tempEmployee.EmpMethod( );

        System.out.println ("Name des Angestellten: " +
                            tempEmployee.employeeName);
    }
}
class Employee {
    public String employeeName;  //employeeName ist public String
    public int employeeNumber;   //employeeNumber ist public integer
    public void EmpMethod( ) {   //EmpMethod ist public Methode
           // hier folgt die Definition der Methode
    }
}
```

Beachten Sie, wie in der Startermethode main() ein Objekt tempEmployee deklariert wird.

Mit Hilfe dieses Objekts wird auf die Members (Data Members : Datenfelder, Attribute; und Function Members: Methoden) des Objekts zugegriffen.

Dies geschieht mit Hilfe der '*qualifizierten Zugriffe'* gemäss folgender Syntax:

reference.member

Beispiel: `tempEmployee.employeeNumber`
oder

reference .method( )

Beispiel: `tempEmployee.EmpMethod( )`

In Java benutzen Sie diese Art des Zugriffs dauernd! Der Zugriff ist nur erlaubt, falls die Grössen als `public` deklariert wurden. `public` ist die Standardeinstellung.

2.6.2.2. Das `private` Keyword

Der andere Extremfall, also der eingeschränkte Zugriff auf die Members, wird als `private` bezeichnet.

Dieses Zugriffsattribut kann vor Variablen oder Methoden stehen. Dadurch kann keine Klasse, kein Objekt und keine Methode von aussen auf diese Methoden oder Datenfelder zugreifen.

Private Methoden werden benutzt, um beispielsweise spezielle Hilfsprogramme zu definieren, welche von ausserhalb des Objekts nicht benutzt und nicht sichtbar sein sollten.

Schauen wir uns ein Beispiel an:

```
class PrivateExample {
    public static void main (String args[ ]) {
        Employee tempEmployee = new Employee( );

        //tempEmployee.employeeNumber = 27;
        //tempEmployee.EmpMethod( );
       System.out.println ("Name des Angestellten: " + tempEmployee.employeeName);

    }
}
class Employee {
    public String employeeName;   //deklariert employeeName als public String
    private int employeeNumber;   //deklariert employeeNumber als einen private Integer
    private void EmpMethod( ) {   //deklariert EmpMethod ist eine private Methode
            // Definition der Methode
    }
}
```

Auch hier wird in der `main()` Methode ein `Employee` Objekt kreiert: `tempEmployee` versucht auf das private Datenfeld (`employeeNumber`) zuzugreifen und die private Methode (`EmpMethod( )`) auszuführen. Beide gehören zur Klasse `class Employee` und sind `private` deklariert.

Die entsprechenden Anweisungen wurden auskommentiert, weil es sich um illegalen Code handelt; private Members darf man nur innerhalb des Objekts verwenden.

Auf der andern Seite ist die Ausgabe von `tempEmployee.employeeName` korrekt, da dieses Datenfeld `employeeName` öffentlich, public, ist.

2.6.2.3. Implementierung von Kapselung

Wie kann man auf private Members zugreifen? Die Antwort ist: indem man eine `public` Methode definiert, welche zur Klasse (und damit zum Objekt) gehört, mit deren Hilfe von aussen auf die privaten Grössen zugegriffen werden kann.

Wir könnten also beispielsweise eine Methode definieren, mit deren Hilfe wir auf das Datenfeld `employeeNum` zugerifen zu können. Die zwei Methoden, zum bestimmen und setzen dieses Werts könnten wir beispielsweise `setEmployeeNumber( )` und `getEmployeeNumber( )` nennen.

```
class PrivateExample {
    public static void main (String args[ ]) {
        EncapsulatedEmployee tempEmployee = new EncapsulatedEmployee( );

        System.out.println ("Angestelltenname: " +
                            tempEmployee.employeeName);
        System.out.println ("Angestelltennummer: " +
                            tempEmployee.getEmployeeNumber ());
    }
}
class EncapsulatedEmployee {
    public String employeeName;
    private int employeeNumber;

    public void setEmployeeNumber(int newValue) {
        employeeNumber = newValue;
    }
    public int getEmployeeNumber( ) {
        return employeeNumber;
    }
}
```

Das Datenfeld ist immer noch private, also nicht allgemein zugänglich - eben ausser für die neu definierten Methoden innerhalb des Objekts (bzw. der Klasse) und von aussen mit Hilfe dieser neu definierten Methoden.

Den Zugriff auf die Methoden selbst können wir ebenfalls noch einschränken, wie gehabt.

2.6.3. Implementierung von Kapselung - Praxis

Nun haben Sie Gelegenheit zu zeigen, was Sie in diesem Modul gelernt haben.

Sie werden den Zugriff auf Methoden und Datenfelder einschränken und mit Hilfe von speziellen Zugriffsmethoden kontrollieren.

1. Im folgenden Programm möchten Sie eine Variable definieren, welche die Postleitzahl der Adresse eines Angestellten enthält. Da diese Information nicht besonders kritisch ist, beschliessen Sie, den Zugriff auf dieses Datenfeld (eine Integer Variable) nicht einzuschränken. [27]

```
public class MeineKlasse {
 public static void main(String args[ ]) {
    ...
```

2. Als nächstes definieren Sie ein Gehaltsfeld.[28]

3. Und nun sollten wir noch eine Methode definieren, welche auf dieses Datenfeld zugreift.[29]

[27] public int empPLTZ; // public kann man auch weglassen
[28] private float salary;
[29] public float getSalary() { return salary; }

2.6.4. Kapselung - Quiz

Mit den folgenden vier Multiple Choice Fragen können Sie Ihr Wissen zu diesem Modul testen.

1. Kapselung ist das Prinzip, mit dessen Hilfe[30]

   a) Daten in andere Methoden geschrieben und aus anderen Methoden gelesen werden.

   b) Hilfsroutinen, für die Manipulation von Datenfeldern definiert und versteckt werden können.

   c) der Zugriff auf die Members eines Objekts öffentlich gemacht wird.

   d) bestimmte Members vor unberechtigtem Zugriff von aussen geschützt werden.

2. Das private Schlüsselwort kann vor Members (Datenfelder, Methoden) gesetzt werden, so dass nur bestimmte Klassen von aussen auf diese Members zugreifen können.[31]

   a) wahr
   b) falsch

3. Welches der folgenden Programmfragmente ist korrekt?[32]

   a) private int employeeNr;
   b) public private float salary;
   c) int private employeeNr;

4. Gegeben sei folgendes Programmfragment

```
public class Employee {
 public int empNr;
 private float empSalary, empTemp;
 private void empMeth() { //...
 }
}
```

   Sie schreiben eine weitere Klasse, in der diese Klasse instanziert wird. Welche der folgenden Programmfragmente sind korrekt?[33]

```
Employee temp = new Employee();
```

   a) `temp.empTemp = 12.23f;`
   b) `temp.empMeth();`
   c) `temp.empNr = 127;`

[30] a)
[31] b)
[32] a)
[33] c)

2.6.5. Kapselung - Zusammenfassung

In diesem Modul haben Sie erste einfache Möglichkeiten kennen gelernt, mit deren Hilfe Sie den Zugriff auf Members einer Klasse einschränken können.

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- den Zugriff auf Members mit Hilfe der Kapselung einzuschränken
- mit Hilfe der Schlüsselworte `public` und `private` die Zugriffsmöglichkeiten auf Members einzuschränken.

2.7. Module 5: Arrays

2.7.1. Einleitung

In diesem Modul lernen Sie komplexere Datenfelder, Arrays, kennen. Diese sind einfache Kollektionen von Variablen des selben Typs. In diesem Modul lernen Sie, wie Sie solche Datenstrukturen manipulieren und beispielsweise auf einzelne Datenfelder zugreifen können.

2.7.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- das Konzept Array zu verstehen
- Arrays zu kreieren, um Basisdatentypen abzuspeichern
- Arrays kreieren, um allgemeine Objekte abzuspeichern
- mehrdimensionale Arrays zu kreieren

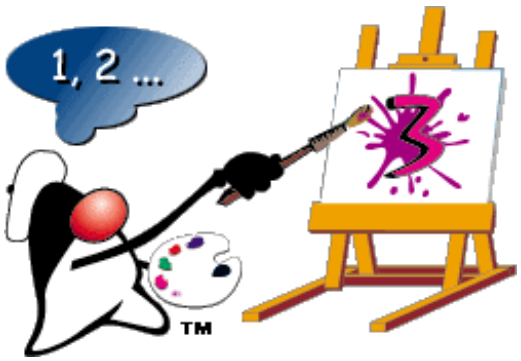
2.7.1.2. Orientieren Sie sich selbst

Als Vorbereitung für diesen Modul sollten Sie sich folgende Fragen stellen:

- wie kann ich eine Gruppe von Objekten des selben Typs zusammenfassen?
- wie kann ich mehrere Objekte des selben Typs effizient manipulieren?
- wie kann ich beispielsweise 100 gleiche Objekte manipulieren?

Falls Sie wissen, wie man mit Arrays umgeht, können Sie diesen Modul überspringen.

Falls Sie diesen Modul durcharbeiten, sollten Sie diese Fragen im Kopf behalten. Sie sollten Ihnen helfen den Text besser zu verstehen.



2.7.2. Was sind Arrays?

Falls Sie beispielsweise die Zahlen von 1 bis 4 abspeichern möchten, könnten Sie dies einzeln tun, jede Zahl in eine Variable. Das mag bei vier Zahlen noch gehen; bei 1000 oder mehr dürfte das etwas schwieriger und mühsamer werden, da Sie immer wieder das selbe tun müssten. Arrays sind spezielle Datenstrukturen und Datenstrukturen hängen mit Algorithmen zusammen. Algorithmen hängen ihrerseits mit Kontrollstrukturen zusammen. Im Falle der Arrays ist die entsprechende Kontrollstruktur die Schleife, speziell die `for` Schleife, da Sie eine bestimmte Anzahl gleicher Elemente in einem ***Array*** verwalten können. Arrays haben wir bereits beim einfachsten Java Programm kennen gelernt: die Kommandozeilenparameter stellen ein `String` - Array dar.

```
class Example {
    public static void main (String args [ ])
{
        int one, two, three, four;
        System.out.println("" + one);
        System.out.println("" + two);
        System.out.println("" + three);
        System.out.println("" + four);
    }
}
```

Das Problem der Verwaltung gleicher Datentypen wächst mit der Anzahl gleicher Elemente. Das einzelne Eintippen jeder Variable wäre bei mehreren hundert Elementen schlicht nicht zumutbar und fehleranfällig.Viele Programmiersprachen bieten Arrays als Speicherform an. Java lehnt sich an die Definition eines Arrays bei C / C++ an. Das System merkt sich die Position der Elemente und vergibt eine sogenannte ***offset Nummer***, eine Platznummer, die in Java mit 0 anfängt. Das erste Element wird also bei der Position 0 abgespeichert.

2.7.2.1. Kreieren eines Arrays

Mit eckigen Klammern wird die Indexierung beschrieben: [ ]. Wie gross ein Array konkret ist und wie es definiert wird, werden wir gleich sehen. In Java ist ein Array ein Objekt, selbst wenn es aus lauter Basisdatentypen besteht. Nun zum Vorgang des Definierens.

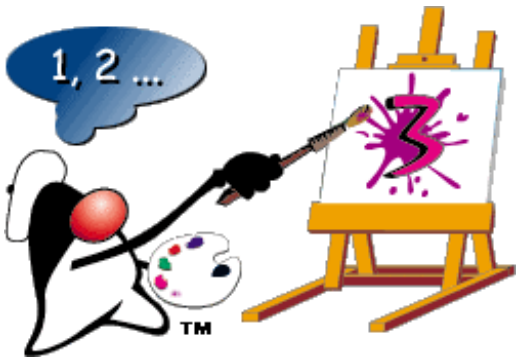
Dieser Prozess ist mehrstufig:

1. Deklaration des Arrays
   Dies definiert lediglich eine Variable, welche das Array Objekt aufnehmen kann.
   Beispiel:
   ```
   Punkt[ ] p;          // ein Array von Punkten
   ```
   oder gleichwertig
   ```
   Punkt p[ ];
   ```

2. Kreieren des Arrays als neues Objekt, also mit dem `new` Operator
   Beispiel:
   ```
   p = new Punkt[5];    // fünf Punkte p[0],p[1],p[2],p[3],p[4]
   ```

3. Speichern der Werte im Array
   Beispiel:
   ```
   p[0]= (12.3,26.2); // Punkt bestehe aus (x,y) Koordinaten
   ```

Fassen wir kurz zusammen:



1. Deklaration des Arrays
2. kreieren des Array Objekts (Instanzierung)
3. Initialisierung der Array Elemente

Einen Sonderfall kennen wir bereits. In der Zeile:

```
public static void main(String[ ] args)
```

definieren wir lediglich, dass die Kommandozeilenargumente in einem Array `args[ ]` abgespeichert sind, wobei jedes Element ein `String` ist.

In diesem Fall brauchen wir das Array Objekt nicht zu instanzieren.

Die Initialisierung von `args[ ]` geschieht nur, falls wir einen Parameter auf der Kommandozeile angeben.

Hier handelt es sich um einen Spezialfall. In der Regel muss ein Array wie oben beschrieben in die Schritten angelegt und initialisiert werden.

2.7.2.2. Instanzierung eines Arrays

Im Beispiel mit den 5 Punkten (`Point p[5]`) sieht die Situation anders aus, eben entsprechend dem Standard.

Im ersten Schritt wird lediglich festgelegt, dass es sich bei den Elementen im Array um Daten vom Datentyp Point handelt.
Im zweiten Schritt wird Speicherplatz angelegt, aber die Slots sind noch leer.
Schritt eins und zwei kann man zusammenfassen, von der Schreibweise her:

```
Point p[ ] = new Point[5];
```

Was haben wir damit erreicht?
Wir haben ein Array Objekt kreiert. Aber nun müssen wir auch noch unsere Elemente kreieren.

Dies geschieht entweder in einer Schleife oder hier zur Veranschaulichung von Hand:
wir müssen fünf Punkt Objekte kreieren und diese jeweils einem Array Element zuweisen.

Dazu wird der Konstruktor der Klasse `Point` aufgerufen, wie wir das bereits von anderen Objekten kennen.

```
p[0] = new Point( );
p[1] = new Point( );
p[2] = new Point( );
p[3] = new Point( );
p[4] = new Point( );
```

Auf die einzelnen Elemente des Arrays kann man zugreifen, falls man die Position im Array kennt. Das erste Element im Array hat die Nummer 0, das letzte also die Nummer 4. Daran muss man sich gewöhnen!

Beim Kreieren des Arrays werden alle Plätze mit einem Standardwert belegt, 0 oder `null` oder dem für den Elementetyp definierten Standardwert.

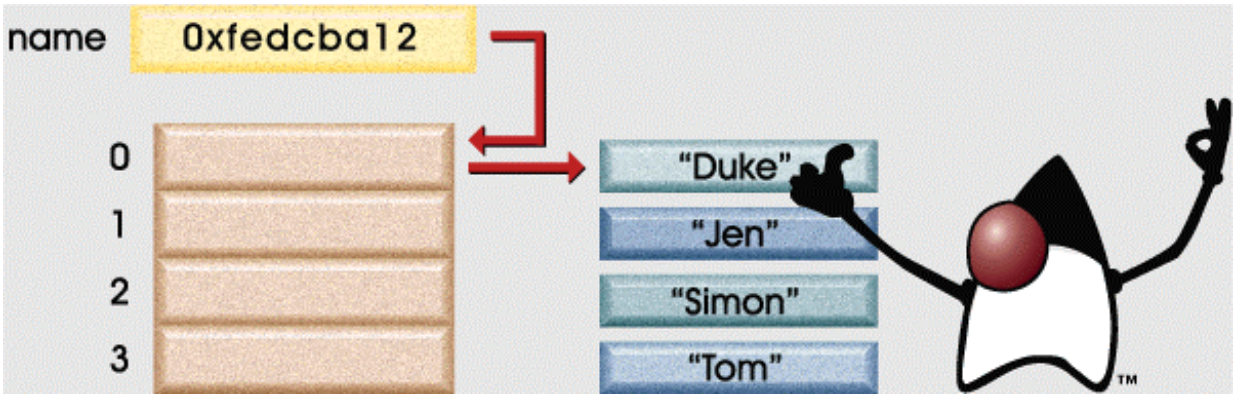
In unserem Fall ist jedes p Element `null`.

2.7.2.3. Eingabe der Werte in ein Array

Die drei Schritte zur Deklaration, Instanzierung und Initialisierung eines Array Objekts kann mana in einen Schritt zusammenfassen, wenigstens scheinbar. Die Array Elemente können gleich initialisiert werden, wie wir das von den Basisdatentypen her kennen.

```
String names[] = {
    "Duke",
    "Jen",
    "Simon",
    "Tom"
};
```

```
String names[];
    names = new String[4];
    names[0] = "Duke";
    names[1] = "Jen";
    names[2] = "Simon";
    names[3] = "Tom";
```



2.7.2.4. Array Grösse und Array Schranken

Die Anzahl Elemente in einem Array wird automatisch als Teil des Array Objekts gespeichert, für den Benutzer jedoch nicht ohne weiteres sichtbar. Dafür stehen jedoch, wie in einem guten objektorientierten System üblich, Methoden zur Verfügung, die diese Informationen liefern. Die Java Virtual Machine (JVM) stellt sicher, dass jeder Zugriff auf ein Element auch korrekt möglich ist, also beispielsweise nicht auf ein Element ausserhalb des Elementebereichs (Indexbereichs) zugegriffen wird, im Beispiel mit den Punkt Elementen beispielsweise auf Element 15 (es wurden lediglich 5 Elemente angelegt). Die JVM wirft eine Ausnahme (Exception: IndexOutOfBoundsException), falls dies trotzdem versucht wird. Exceptions sind Laufzeitfehler, die im Programm abgefangen werden müssen. Falls Sie in Ihrem Programm nichts dagegen tun, stürzt das Programm ab. Sie entsprechen in etwa dem, was man als *Interrupt* bezeichnet.

Schauen wir uns ein **Beispiel** an:

```
String [ ] names = new String [10];
```

Nun versuchen wir einem Element den Wert `Jane` zuzuordnen. Sie sehen das Fragemnet des programmcodes unten. Diese Zeile würde eine Exception auslösen, weil die Elemente von 0 bis 9*, also 10 nicht*, angelegt wurden. 10 wäre also das 11te Element und somit ausserhalb der Arraygrenzen:

```
names[10] = "Jane";
```

Bemerkung - Die 10 im ersten Programmcode Fragment bezieht sich auf die Anzahl Elemente, die das Arrayobjekt `names` aufnehmen kann. Die 10 im zweiten Programmcode Fragment bezieht sich auf ein Element, eine Referenz, dem wir einen Wert zuweisen möchten.

2.7.2.5. Multidimensionale Arrays

Alle bisher verwendeten Arrays sind eindimensional - also Listen von Werten oder Elementen oder Objekten. Es ist aber auch möglich die Anzahl Dimensionen zu erhöhen, beispielsweise um zweidimensionale Datenstrukturen (Ebenen, Schachbretter, Tabellen) zu beschreiben oderdreidimensional (Raum, irgendwelche Auswertungen). Mehrere Dimensionen werden durch mehrere Klammerpaare beschrieben.

Im folgenden Beispiel deklariert die erste Anweisung ein Array von 12 Integer Variablen und instanziert sie (initialisiert sie aber nicht: alle haben den Standardwert 0). Ein solches Element könnte beispielsweise die monatliche Durchschnittstemperatur oder den Niederschlagswert eines Monats abspeichern.

```
int [ ] oneYear = new int[12];
int [ ][ ] fiveYears = new int[5][12];
```

Die zweite Anweisung deklariert und instanziert ein zweidimensionales Array, um beispielsweise die Werte aus der ersten Zeile über 5 Jahre abzuspeichern. Insgesamt kann man in diesem Array also 5 * 12 = 60 Werte abspeichern.

Der Zugriff auf einzelne Array Elemente geschieht analog zum Zugriff auf eindimensionale Arrays. Nur müssen wir jetzt einfach zwei Indices angeben. Wichtig ist, dass die Variablen sinnvolle Namen (und die Indices ebenso) haben. Sonst wird die Lesbarkeit der Programme erschwert.

Die Anzahl Elemente in irgend einer Dimension wird mit der Standardmethode für Array Objekte `length` bestimmt.

Im **Beispiel** unten hat `fiveYears.length` den Wert 5 und `fiveYears[yr].length` hat den Wert 12:

```
int yr, mnth;
for (yr=0 ; yr < fiveYears.length ; yr++) {
    for (mnth=0 ; mnth < fiveyears[yr].length ; mnth++) {
        System.out.println("" + fiveYears[yr][mnth]);
    }
}
```

In diesem Fall hat jedes Array fiveYears[yr][] 12 Elemente; jedes Array fiveYears[][mnth] 5. Mehrdimensionale Arrays kann man ebenfalls gleich bei der Definition initialisieren. Einzig die Reihenfolge muss man beachten!

Im folgende Beipiel wird eine Dreieck Konstruktion definiert, analog zum Pascal'schen Dreieck:

```
int a[][] = { {0},  {1,2},  {3,4,5},  {6,7,8,9} };
```

Die einzelnen Arrays sind in der äusseren Klammer klar erkennbar.

2.7.2.6. Arrays bestehend aus Objekten

Arrays können auch Objektreferenzen enthalten. Falls die Klasse Cat existiert, kann eine Liste mit Katzen folgendermassen angelegt werden:

```
int i;                  //eine int Variable
Cat b;                  //eine Cat Klassenreferenz (Objekt)
int[] ia;               //eine Referenz auf ein Array von Integern
Cat[] ba;               //ein Array von Cat Referenzen
```

Die Objekte selbst sind damit natürlich noch nicht angelegt!
Falls Sie auch das noch wollen, müssen Sie das programm wie unten ergänzen.

```
Cat [] ba = new Cat [3];        //Array für 3 Katzen
ba[0] = new Cat();
ba[1] = new Cat();
ba[2] = new Cat();
```

Analog verhält es sich bei Basisdatentypen. Im folgenden Beispiel wird ein Array Objekt für Integer Werte angelegt. die Variablen selbst haben alle den Standardwert 0:

```
int [] ia = new int [10];       //Array für 10 int Variablen
```

Alle Methoden und Eigenschaften, die wir für Basisdatentypen Arrays besprochen haben, gelten auch für Objekt- Arrays.

2.7.3. Arrays - Praxis

Jetzt liegt es wieder bei Ihnen: Sie sollten das Gelesene und Gelernte praktisch anwenden und Ihr Wissen festigen. Wie auf dem Bild unten, sollten Sie also (wie Duke, so heisst das Kerlchen), etwas Gymnastik machen:

Die Übungen verlaufen wie die bisherigen. Sie können aus mehreren Antworten auswählen oder Programmcode ergänzen.

1. Die folgende Klassendefinition muss ergänzt werden und zwar so, dass zwei Hunde kreiert werden (deklariert und instanziert), die Klasse `Dogs` wurde bereits erstellt:

```
class HundeWetter {
   public static void main(String args[ ]) {
          ...[34]
```

2. Jetzt müssen Sie die zwei Hunde auch noch kreieren.

```
class HundeWetter {
   public static void main(String args[ ]) {
          Dogs hund[ ] = new Dogs[2];
          ...[35]
          ...
          ...
   }
}
```

[34] Dogs hund[ ] = new Dogs[2];

[35] hund[0] = new Dogs();
hund[1] = new Dogs();

2.7.4. Arrays - Quiz

Im folgenden Quiz können Sie Ihr Wissen über Arrays festigen.

Sie sollten nach dem Durcharbeiten dieses Moduls in der Lage sein, die Antworten auf die Fragen geben zu können.

Sie sollten aber zusätzlich praktische Übungen machen, am Rechner, in der von Ihnen gewünschten Entwicklungsumgebung.

1. Welche der folgenden Array Deklarationen ist korrekt?[36]
   a) `int [ ] buglist = {"Fruchtfliege", "Schabe", "Nachtfalter"};`
   b) `String partyliste[ ] = {"Mike", "Britta", "Jens", "Nicole"};`
   c) `char[ ] zeichenliste = {'a','c','ü','o'}`
   d) `int[ ] punkte = {1.2, 2.1, 4.3, 8.5, 22, 3, 7};`

2. Gegeben sei eine Zeichenliste, welche 'g' enthält:
   `char[ ] zeichenListe = {'a', 'b', 'g', 'i'};`
   Welche der folgenden Anweisungen verwendet 'g'?[37]
   a) `meinChar = zeichenListe[0];`
   b) `System.out.println(zeichenListe[1]);`
   c) `meinChar = zeichenListe[2];`
   d) `meinChar = zeichenListe[3];`
   e) `meinChar = zeichenListe[4];`

3. Gegeben sei folgende Deklaration:
   `int kundenNummer[ ] = new int[1000];`
   Welche der folgenden Anweisungen ist korrekt?

   a) `kundenNummer[0] = 102;`
   b) `kundenNummer[213] = 1.25;`
   c) `kundenNummer[1020] = 223;`

4. Gegeben sei folgende Deklaration:
   `int[ ] liste = new int[1000];`
   Welche der folgenden Anweisungen ist nicht korrekt?[38]

   a) list[0]=1010;
   b) list{999} = -10;
   c) list[567] = 22;

[36] b) : bei den andern stimmt der Datentyp nicht mit den Elementen überein bzw. das Semikolon fehlt (char[])
[37] c) : das dritte Element ist 'g', also ...[2], weil bei 0 begonnen wird.
[38] b) die falschen Klammern wurden verwendet: { } statt [ ]

2.7.5. Zusammenfassung - Arrays

In diesem Modul haben Sie die Datenstruktur Array und deren Implementation in Java kennen gelernt.

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- das Konzept eines Arrays zu verstehen
- Arrays einsetzen können, um Basisdatentypen zu speichern
- Arrays einsetzen können, um Objekte abspeichern zu können
- mehrdimensionale Arrays definieren und einsetzen können.

2.8. Module 6: Fortgeschrittene Objekt-Orientierte Konzepte

2.8.1. Einführung in Fortgeschrittene objektorientierte Konzepte

Dieser Modul führt Sie ein in fortgeschrittene objektorientierte Konzepte, die Sie in der Java Programmierung einsetzen können.

2.8.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- Konstruktoren einzusetzen, um Objekte kontrollierter zu kreieren.
- Vererbung einzusetzen, um Teile bestehender Klassendefinitionen weiterverwenden zu können.
- abstrakte einzusetzen, um Ihre Objekte besser organisieren zu können.
- Polymorphismus einzusetzen, um generische Referenzvariablen zu kreieren und zu verwenden.

2.8.1.2. Orientieren Sie sich selbst

Sie kennen das Prozedere bereits: vor dem Durcharbeiten eines Moduls sollten Sie sich zuerst "warm laufen".

Überlegen Sie sich folgendes und stellen Sie sich folgende Fragen:

- kennen Sie die OO Konzepte bereits ? Vererbung? Konstruktoren? Polymorphismus?
- wie beeinflusst Vererbung, auf Grund der Definition, Klassen?
- was heisst das: "Programmcode wird vererbt"?

Falls Ihnen die Konzepte vertraut sind, können Sie im Quiz, am Ende des Moduls überprüfen, wie gut Sie sind. Die Fragen sind bewusst einfach gehalten, um Ihnen Erfolgserlebnisse zu geben und Sie zu selbstständigem Arbeiten anzuregen.

2.8.1.3. Kurze Zusammenstellung der wichtigsten OO Konzepte

Die folgenden objektorientierten Programmierkonzepte werden in diesem Modul besprochen, auf die Art und Weise, wie sie in Java eingesetzt werden.

- **Konstruktoren -**
  Gestatten es Ihnen neue Objekte zu kreieren, in der Regel mit definierten Anfangszuständen.

- **Vererbung -**
  Dieses Prinzip, nach dem eine neue Klasse oder ein neues Objekt Variablen und Methoden anderer Klassen und Objekte übernehmen kann, lässt sich mit Hilfe der Frage "is-a testen:
  wenn Sie zwei Klassen oder Objekte haben (Katze und Lebewesen zum Beispiel) können Sie sich fragen, ob ein Objekt der einen Klasse auch ein Objekt der andern Klasse ist (eine Katze ist auch ein Lebewesen).

  Dann ist die eine Klasse die Oberklasse, die andere die Unterklasse, immer relativ gesehen (die Klasse Lebewesen ist die Oberklasse zur Katzenklasse; die Katzenklasse ist die Unterklasse zur Klasse Lebewesen).

- **Containment (Eingrenzung) -**
  Man kann neue Klassen bilden, indem man Variablen umd Methoden einer bestehenden Klasse übernimmt. Der Einsatz der Methoden und Datenfelder in der neuen Klasse ist allerdings spezifischer.

  Man spricht von einer "has-a" Beziehung, weil Sie die Frage stellen können:
  ist X in Y enthalten?

  Beispiel: eine Küche enthält normalerweise einen Herd, Kühlschrank, ...

- **Abstract Class (abstrakte Klassen) -**
  Sie können Klassen definieren, die keinen oder kaum konkreten Programmcode enthalten, aber bereits definieren, welche Methoden (und Datenfelder) in der vollständigen Klasse vorhanden sein werden. Abstrakte Klassen haben also so etwas wie eine Platzhalterfunktion. Sie müssen später erweitert und dadurch implementiert werden, sonst sind sie unbrauchbar!

  Beispiel: Sie könnten eine abstrakte Klasse Fahrzeg definieren und Anschliessend daraus die Klassen Motorfahrzeug, Motorrad, ... herleiten bzw. durch zusätzliche Attribute und Methoden diese Klassen definieren. Alles, was Motorfahrzeugen und Motorrädern gemeinsam ist, packen Sie in die (abstrakte) Klasse Fahrzeug.

- **Polymorphismus -**
  Das Prinzip, welches es dem Programmierer gestattet, Operationen einzusetzen, ohne sich zu sehr Gedanken über die verwendeteten Datentypen zu machen (mehrfache Darstellung, Erscheinungsform von einem und demselben).

2.8.2. Konstruktoren - Was sind Konstruktoren?

Konstruktoren sind spezielle Methoden, mit denen Sie das Kreieren neuer Objekte besser kontrollieren können. Bei der Instanzierung wird der Konstruktor jedesmal aufgerufen, auch wenn Sie selbst keinen Konstruktor definiert haben (in diesem Fall wird ein Default Konstruktor aufgerufen).

Im folgenden Beispiel wird wieder einmal eine Katze kreiert, als Instanz der Klasse Katze:

```
Cat duncan = new Cat( );
```

Konstruktoren verhalten sich, obschon es sich um spezielle Methoden handelt, wie allgemeine Methoden, mit folgenden Ausnahmen:

- Konstruktoren haben keinen Rückgabetyp (return type)
- der Name des Konstruktors ist immer identisch mit em Klassennahmen (wird also gross geschrieben, entgegen der Namensgebung für Methoden).

Und hier ein Beispiel, wieder für die Katzenklasse:

```
class Cat {
    String name;
        // Definition des Konstruktors
        public Cat( ) {
            name = "Duncan";
        }
}
```

Dieser Konstruktor macht nicht besonders viel und ist alles andere als clever: er setzt den Namen der Katze einfach auf "Duncan", also eine Zeichenkette, welche konstant ist.

Besser wäre ein Konstruktor, bei dem der Namen der Katze übergeben wird, als Parameter. Darauf kommen wir gleich noch!

Sie können gleichzeitig mehrere Konstruktoren definieren. Das System sucht sich dann jeweils den passenden aus. Im folgenden Beispiel überschreiben Sie den Default Konstruktor (ohne Argumente) und definieren einen Konstruktor mit einer Zeichenkette als Parameter. Damit können Sie natürlich keinen weiteren Konstruktor mit einer Zeichenkette defeinieren, beispielsweise mit dem Vornamen als Parameter. Pro Paramneterliste können Sie also genau einen Konstruktor definieren (man sagt auch: pro Signatur):

```
class Cat {
    String name;
        // der Konstruktor
        public Cat( ) {
            name = "Duncan";
        }
        // noch ein Konstruktor
        public Cat(String inName) {
            name = inName;
        }
}
```

Im obigen Beispiel wird der Konstruktor überladen ("overloading"). Dies ist gestattet, weil die zwei Methoden / Konstruktoren unterschiedliche Parameterlisten besitzen.

Nn können wir Katzen auf mehrere Arten kreieren:

```
Cat duncan = new Cat( );
Cat kevin = new Cat("Kevin");
```

Im ersten Fall kreieren Sie eine Katze mit dem Namen "Duncan", weil Sie im Default Konstruktor diesen Namen als Standard definiert haben.

Kevin dagegen wird mit einem andern Konstruktor kreiert. Er erhält seinen Namen als Parameter des Konstruktors.

2.8.2.1. Return Type

Normale Methoden müssen den Datentyp des Rückgabewertes angeben. Dies gehört zur Methodendeklaration. Falls kein Wert zurück gegeben wird, muss `void` angegeben werden.

Der Typ des Rückgabewertes wird vom Compiler geprüft!

Der Konstruktor ist aber eine spezielle Methode - er liefert ja eigentlich eine Instanz der Klasse, zu der er gehört.

2.8.2.2. Der Default Konstruktor

Wir haben ihn schon öfters erwähnt, den Default Konstruktor: in Java muss jede Klasse mindestens einen Konstruktor besitzen. Falls der Compiler eine Klasse ohne Konstruktor findet, fügt er einen Standardkonstruktor hinzu. Dieser Konstruktor tut eigentlich nichts; aber er sorgt dafür dass die Objekte konstruiert werden können.

Dieser Konstruktor heisst ***Default Konstruktor***.

```
class Dog {
    String name;
}
```

In dieser Klassendefinition fehlt ein Konstruktor. Der Compiler fügt einen hinzu. Damit sieht die Klasse gleich aus, wie wenn Sie folgende Klassendefinition getätigt hätten:

```
class Dog {
    String name;
    // default Konstruktor
    public Dog() {
    }
}
```

Bemerkung - Sobald Sie irgend einen Konstruktor definieren, sei es auch nur einen leeren wie im obigen Beispiel, lässt der Compiler die Klasse in Ruhe!

2.8.3. Konstruktoren - Praxis

Diese Übung dient der Vertiefung des Konstruktorkonzepts.

Sie erhalten wie immer bestimmte Antworten als mögliche Lösungen.
Wählen Sie die richtige aus oder schauen Sie die Musterlösung in der Fussnote nach.

1. Gegeben sei die Klasse Gecko, wie unten vorgegeben. Schreiben Sie einen Default Konstruktor für diese Klasse.[39]
   ```
   class Gecko {
     ...
   }
   ```

2. Ergänzen Sie nun Ihre Klasse durch eien Konstruktor, der eine Zeichenkette als Parameter akzeptiert (aber keiner Variable zuweist: es handelt sich um eine reine Übungsaufgabe). Die Parametervariable heisst `skinColor`:[40]
   ...
   ...
   ...
   ...
   ...
   ...

[39] public Gecko() { }
[40] public Gecko(String skinColor) { }

2.8.4. Vererbung - Was ist Vererbung?

Vererbung (engl *Inheritance)* ist die OO Bezeichnung für das Zusammenfassen mehrerer Klassen auf Grund gemeinsamer Merkmale (Methoden, Attribute).

Sie kennen viele solche Systeme / Situationen aus dem Alltag: ein Manager und eine Sekretärin sind beides Angestellt.

```
class Manager {
int employeeNumber;
String name;
int departmentNumber;
int extensionNumber;
int salary;
int numberOfWorkers;
// und so weiter
}
```

```
class Secretary {
int employeeNumber;
String name;
int departmentNumber;
int extensionNumber;
int salary;
Manager worksFor;
// und so weiter
}
```

Vergleichen Sie den programmcode. Sie werden sehen, dass eine bestimmte Ähnlichkeit vorhanden ist. Wenn Sie nun beide Klassen weiter entwickeln und pflegen wollen, benötigen Sie viel Geduld und in absehbarer Zeit werden Sie viel Arbeit haben, weil Sie zwei Klassendefinitionen unterhalten müssen, die sehr ähnlich sind.

Sie sollten Also versuchen, die beiden Definitionen zusammen zu führen. Sie können dies tun, indem Sie eine Oberklasse definieren und aus dieser die Manager und Sekretärinnen Klasse herleiten, durch Vererbung.

2.8.5. Die "is-a" Beziehung

Vererbung ist ein wichtiges Hilfsmittel, mit dem Sie in OO Sprachen eine oder mehrere Klassen erweitern können und alle Methoden und Datenfelder aus der Oberklasse weiterverwenden können.

In Java wurden die Möglichkeiten etwas eingeschränkt: Sie können eine Klasse aus höchstens einer Oberklasse herleiten, man spricht auch von Einfachvererbung. Andere Sprachen erlauben auch mehrere Oberklassen, Mehrfachvererbung. Allerdings kann dies zu Problemen führen!

verebung sollten Sie jedoch nur dann einsetzen, wenn es Sinn macht. Ob es Sinn macht können Sie mit der Frage "**is-a**" prüfen, man spricht von einer "is-a" Beziehung.

Beispielsweise: der Manager *is-a* Angestellter; die sekretärin *is-a* Angestellter.Es macht also Sinn, den Angestellten als gemeinsame Oberklasse zu definieren.

Schauen wir uns ein Beispiel an:

```
class Cycle {
    int numberOFWheels;
    int numberOfSeats;
    float luggageCapacity;
    // ...
}
class Boeing747 {
    int numberOFWheels;
    int numberOfSeats;
    float luggageCapacity;
    int numberOfWings;
    // ...
}
```

Programmcodeähnlichkeit reicht jedoch nicht aus, um Klassen zu abstrahieren. Sie sollten vorsichtig mit der Vererbung umgehen. Vererbung kreiert Overhead, benötigt Ressourcen und macht damit Programm, falls falsch eingesetzt, langsamer.

2.8.6. Der "is-a" Beziehungstest

Wir sehen auf den ersten Blick Gemeinsamkeiten und versuchen eine gemeinsame Klasse zu definieren.

```
class Cycle {
    int numberOFWheels;
    int numberOfSeats;
    float luggageCapacity;
    // ...
}
class Boeing747 extends Cycle {
    int numberOfWings;
    // ...
}
```

Auf den ersten Blick mag das ganz toll aussehen. Sie sind ein clevers Bürschchen!
Aber nun machen wir den "is-a" test: "eine Boeing 747 ist ein Fahrrad"

Klingt irgendwie komisch; Sie hätten auch Probleme mit einer 747 auf dem Fahrradweg!

Falls die Aussage keinen Sinn macht, dann behält man die Redundanz bei! Aber Sie haben noch weitere Versuche. Beispielsweise könnten Sie ein generelles Beförderungsmittel oder ein "Fahrzeug" definieren.

```
class Vehicle {
    int numberOFWheels;
    int numberOfSeats;
    float luggageCapacity;
    // ...
}
class Cycle extends Vehicle {
    // ...
}
class Boeing747 extends Vehicle {
    int numberOfWings;
    // ...
}
```

Das wäre eine Möglichkeit. Aber falls Sie in Ihren Programmen nur Flugzeuge behandeln wollen, macht auch dies keinen Sinn. Sonst kann es eine brauchbare Lösung sein.

2.8.7. Containment - Enthaltensein

Sie haben auch noch weitere Möglichkeiten, Klassen zu kombinieren! Eine davon lernen wir nun kennen.

Betrachten Sie eine Küche. Diese enthält verschiedene Geräte, Kühlschrank, Geschirrspühler und vieles mehr.

Nun versuchen wir mit Hilfe von Vererbung diese unterschiedlichen Klassen unter einen Hut zu bringen.

```
class Cooker {
    //...
}
class Refrigerator {
    // ...
}
class Kitchen extends Cooker, Refrigerator {
    // ...
}
```

Das Beispiel scheitert schon an einer grundsätzlichen Einschränkung von Java: es ist nicht möglich eine Klasse aus mehreren anderen Klassen herzuleiten (Mehrfachvererbung).

Auch der "is-a" Test sieht schlecht aus: die Küche ist kein Kühlschrank, keine Geschirrspühlmaschine oder elektrisches Brotmesser.

Alternativ können wir die Klasse Küche aus Bestandteilen zusammen bauen, wie der Schreiner!

Dies geschieht, indem wir in der Klasse Küche Instanzen der andern Klassen bilden. Die einzelnen Klassen bleiben unabhängig. Kühlschränke können Sie ja auch sonstwo einsetzen.

```
class Cooker {
    //...
}
class Refrigerator {
    // ...
}
class Kitchen {
    Cooker myCooker;
    Refrigerator myRefrigerator;
    // ...
}
```

Wir könnten aber eine Klasse Raum definieren und die Küche diese Klasse erweitern lassen. Das macht mehr Sinn und erfüllt eine sinnvolle "is-a" Beziehung.

2.8.8. Die "has-a" Beziehung

Das Containment, das Enthaltensein von Objekten in andern, wie oben bei der Küche, kann man mit einer andern Frage leicht testen.

Wie die "is-a" Frage die Vererbung überprüft, prüft die "**has-a**" Frage das ***Enthaltensein***, das Containment von Klassen / Objekten in anderen.

Und hier ein Beispiel:

```
class Cooker {
    //...
}
class Refrigerator {
    // ...
}
class Kitchen {
    Cooker myCooker;
    Refrigerator myRefrigerator;
    // ...
}
```

Die Frage: "hat meine Küche einen Herd" ist eine gute Frage. Die has-a Frage macht also Sinn. Wir haben also ein Containment vor uns!

2.8.9. Vererbung - Praxis

Und wieder sollten Sie Ihr Wissen festigen und einige praktische Übungen machen.

1. Sie haben eine Klasse Gecko, die Sie nun noch erweitern möchten. Da Biologie nicht meine Spezialität ist, nennen wir die Unterklasse einfach Becko. Konstruieren Sie diese Klasse.

```
class Gecko {
 public Gecko() { }
}

class Becko ....[41]
```

2. Aber es gibt auch noch Heckos, die eine Untergattung der Beckos sind.
   Wie sieht denn diese Klass aus?

```
class Gecko {
    public Gecko() { }
}
class Becko extends Gecko {
   public Becko() { }
}
class Hecko ...[42]
```

Nun können Sie schon bald Ihren Stammbaum programmieren!

[41] public class Becko extends Gecko {
  public Becko() { }
}

[42] public class Hecko extends Becko {
 public Hecko() { }
}

2.8.10. Abstrakte Klassen

Sie können auch eine Klasse definieren, welche eigentlich fastnichts tut oder kann.

Betrachten wir folgende Klassenhierarchie:

```
class Mammal {
    // was haben alle Säugetier gemeinsam?
}
class Dog extends Mammal {
    // was haben Hunde gemeinsam?
}
class Cat extends Mammal {
    // was haben Katzen gemeinsam?
}
class Rabbit extends Mammal {
    // was haben Kaninchen gemeinsam
}
```

Würden Sie in einer Haustierhandlung fragen, ob der Laden auch Säugetiere hat?

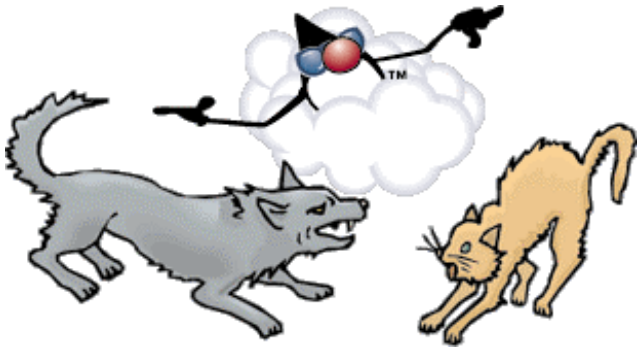
Sie brauchen keine Instanzen dieser (abstrakten) Klasse. Aber die Klasse leiset gute Dienste: sie fast Gemeinsamkeiten zusammen!

In Java können Sie Klassen mit dem Attribut abstract versehen, um diese Fähigkeit zu erhalten, die wir gerade besprochen haben.

```
abstract class Mammal {
    // ... alle Säugetiere haben ... sind... können...
}
```

2.8.11. Polymorphismen

Polymorphismus ist ein Wort aus dem Griechischen und besagt soviel wie "mehrere Formen".



In OO wird es benutzt, um auszudrücken, dass eine Klasse unterschiedlich erweitert werden kann. Jeder der Unterklassen kann an Stelle der Oberklasse verwendet werden (sie ist eine Verfeinerung der Oberklasse; "is-a"). Die Oberklasse hat unterschiedliche Formen, je nach Unterklasse!

Mit Polymorphismus kann man eine Referenzvariable einer Oberklasse benutzen, um dort wo ein Objekt der Unterklasse verlangt wird (die Unterklasse liefert ja Objekte, die auch Oberklassen-Objekte sind, wegen der Vererbung: ein Hund als Objekt ist auch ein Objekt als Säugetier).

Diesen Gedanken können wir jetzt weiterführen und auch abstrakte Klassen mit in die Überlegungen einbeziehen. An Stelle einer konkreten Klasse könnten wir also auch eine abstrakte einsetzen : das (abstrakte) Säugetiert kann Platzhalter für eine Kuh sein (eine Implementierung der abstrakten Klasse Säugetier), selbst wenn die abstrakte Klasse nicht instanziert werden kann.

```
abstract class Mammal {
    // Säugetiervariablen
}
class Dog extends Mammal {
    // Variable für Hunde
}
class Cat extends Mammal {
    // Variablen für Katzen
}
class PolymorphicExample {
    public static void main (String args[]) {
        Mammal m1 = new Dog(); // Hund ist auch Mammal
        Mammal m2 = new Cat(); // Katze ist auch Mammal
    }
}
```

2.8.11.1. Polymorphklasse Parameter

Falls Sie eine Methode haben, welche einen bestimmten Objekttyp (Objekte einer bestimmten Klasse) als Parameter erwarten, dann können Sie an Stelle dieses Typs, eines Objekts dieser Klasse, auch ein Objekt einer Oberklasse verwenden.

```
class Vet {
    void vaccinate (Mammal m) {
        // vaccinate m
    }
}
abstract class Mammal {
    // Mammal members
}
class Dog extends Mammal {
    // Dog members
}
class Cat extends mammal {
    //Cat members
}
class Example {
    public static void main (String args[]) {
        Vet doctor = new Vet();
        Dog myDog = new Dog();
        Cat myCat = new Cat();
        doctor.vaccinate(myDog);
        // Parameter (myDog) muss Mammal sein und ist es
        doctor.vaccinate(myCat);
        // Parameter (myDog) muss Mammal sein und ist es
    }
}
```

Die Methode `vaccinate` erwartete als Parameter ein Säugetier, `Mammal`; aber wir lieferten lediglich Objekte der Unterklassen, `myDog` und `myCat` - beides sind Säugetiere, `Mammals` weil beide Klassen `Dog` und class `Cat` die Klasse `Mammal` eweitern. Damit haben wir mehr Flexibilität und brauchen die Methode `vaccinate` nicht zu überladen, also mehrfach zu definieren (einmal für Hunde, einmal für Katzen, einmal für ...), für jede denkbare Unterklasse der Säugetierklasse `Mammal`.

2.8.11.2. Polymorphe Kollektion

Sie können in einem Array lediglich einen Datentyp speichern, eine Art Objekte. Ein Array von Integers kann lediglich Integers aufnehmen. Falls wir ein Array von Objekten kreieren, dann können wir neben dem bestimmten Objekttyp, für den wir das Array kreieren, auch Objekte der Unterklassen einsetzen, da sie ja auch Objekte der Oberklasse sind (der Hund ist ein Objekt der Klasse Hund, aber auch ein Objekt der Klasse Säugetiere).

Falls wir also `Cat`, `Dog` und `Mammal` Klassen unabhängig definiert haben, ist folgendes Programmfragment syntaktisch korrekt:

```
Cat [ ] catArray = new Cat [3];
catArray[0] = new Cat( );
catArray[1] = new Cat( );
catArray[2] = new Cat( );
```

In diesem Fall dürfen wir an Stelle einer Katze auf keinen Fall einen Hund verwenden. Das Array ist genau für Katzen gemacht und nur für Katzen (oder eventuell noch spezielle Unterklassen der Katzen).

Auf der andern Seite könnten wir auch ein Array von Säugetieren definiern. Dann hätten wir dieses Problem nicht:

```
Mammal [ ] mammalArray = new Mammal [3];
mammalArray[0] = new Cat( );
mammalArray[1] = new Dog( );
MammalArray[2] = new Cat( );
```

Nun können wir alle Säugetiere, die der Veterinär in seinen Zwingern hat, in einem gemeinsamen Array verwalten.

2.8.12. Quiz - Polymorphismus

Zuer Abwechslung noch etwas zur Festigung Ihres Wissens. Sie sollten jetzt Grundkenntnisse über Polymorphismus haben. Wie gut Ihr wissen ist, können Sie gleich testen.

1. Was wird bei der Instanzierung jedesmal aufgerufen?[43]

   a) Destruktor
   b) eine Methoden Deklaration
   c) Konstruktor
   d) eine Typendeklaration

2. Welcher Begriff beschreibt das Zusammenfassen mehrerer Objekte zu einem, basierend auf beispielweise einer gemeinsamen Aufgabe, eines gemeinsamen Themas?[44]

   a) Inheritance - Vererbung
   b) Containment - Enthaltensein
   c) Polymorphismus
   d) Abstraktion

3. Klingt die folgende Klassenerweiterung wie eine "is-a" Beziehung?[45]

   ```
   class Fahrzeug extends Computer
   ```

   a) nein
   b) ja

4. Der Rückgabewert eines Konstruktors muss ein Basisdatentyp sein[46]

   a) ja
   b) nein

5. Das folgende Schlüsselwort wird eingesetzt um eine Klasse zu kennzeichnen, die nie instanziert werden soll:[47]

   a) final
   b) private
   c) polymorphic
   d) abstract
   e) protected

[43] c) Konstruktor
[44] a) Vererbung
[45] a) nein
[46] b) nein
[47] d) abstract

2.8.13. Zusammenfassung - Fortgeschrittene OO Konzepte

In diesem Modul haben Sie mehrere Konzepte kennen gelernt, welche Sie in der Java Programmierung anwenden können.

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- Konstruktoren einzusetzen, um Objekte kontrolliert zu kreieren
- Vererbung einzusetzen, um Teile einerKlassendefinition wieder zu verwenden.
- abstrakte Klassen einzusetzen, um Objekte zu organisieren
- Polymorphismus einzusetzen, um generische Referenzvariablen zu kreieren.

2.9. Kurs Zusammenfassung

Sie haben diesen Kursteil abgeschlossen.

Herzliche Gratulation!

Nun bleibt Ihnen noch einiges zu tun, bis Sie ein professioneller Java Programmierer sind. Aber Sie sind auf dem besten Weg.

Vergessen Sie eines nicht: Programmieren ist eine Aktivität, die man also ausüben muss. Programmieren lernen Sie nicht aus einem Buch. Sie lernen es durch Praxis.

PROGRAMMIEREN MIT JAVA