252-0027 Merged

Version: 1.2

Author: Ruben Schenk

Date: 12.01.2020

Changelog:

1.2: Fixed spacing in 5.X

1.1: Fixed spacingin 7.3

1.0: Initial release

1. EBNF Notation

1.1 EBNF Regeln und Beschreibungen

Eine **EBNF-Beschreibung** besteht aus einer Menge von *EBNF-regeln*.

Eine EBNF-Regel sieht wie folgt aus:

```
LHS <== RHS
```

Wobei `LHS` ein Wort in *kursiv* ist, welches den Namen der EBNF-Regel bezeichnet. `RHS` bildet dann die genaue Beschreibung für den Namen (LHS).

Die EBNF-Notation bietet **vier control forms** zum erstellen von EBNF-Beschreibungen. Diese sind:

- Aufreihung ("sequence")
- Entscheidung ("decision")
- (Auswahl)
- Wiederholung ("repetition")
- Rekursion ("recursion")

1.1.1 Aufreihung ("sequence")

```
LHS <== form_1  form_2  form_3
Beispiel: <digit_123> <== 1  2  3
```

Dabei wird die Beschreibung von links nach rechts gelesen und die Reihenfolge ist wichtig.

1.1.2 Auswahl

```
LHS <== form_1  |  form_2
Beispiel: <raum> <== E12  |  D28
```

Die Auswahl stellt eine Menge von Alternativen dar, wobei die Reihenfolge unwichtig ist und die einzelnen Alternativen durch einen stroke | getrennt sind.

1.1.3 Option/Entscheidung ("decision")

```
LHS <== [form_1]
Beispiel: <vorzeichen> <== [ + | - ]
```

Elemente in einer Option werden in square brackets getan, diese Option kann gewählt werden, *muss aber nicht gewählt werden.*

1.1.4 Wiederholungen ("repetitions")

```
LHS <== { form_1 }
Beispiel: *digit_sequence* <== { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 }
```

Elemente ein einer Wiederholung können so oft wiederholt werden wie sie wollen, das heisst, auch *null Wiederholungen sind möglich!*

1.1.5 Rekursion ("recursion")

```
LHS <== form_1 | LHS
Beispiel: <balance> <==  | (A <balance> B)
```

Eine EBNF-Regel ist direkt rekursiv, wenn ihr Name in der Definition verwendet wird. Eine rekursive Beschreibung ist manchmal nötig um komplizierte Symbole zu beschreiben.

Beispiel EBNF-Beschreibung *integer:*

```
<sign> <== + | -
<digit> <== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<integer> <== [ <sign> ] <digit> { <digit> }
```

1.2 Beweisen, dass ein Symbol zu einer Beschreibung passt

Folgende Beweismöglichkeiten sind verfügbar um die Legalität eines Symboles zu zeigen:

- *Tabellen*: Jede Zeile wird aus der Vorgängerzeile durch eine Regel abgeleitet
- *Ableitungsbäume*: Graphische Darstellung eines Beweises durch eine Tabelle

2. Einfache Java Programme

2.1 Einführung

Ein einfaches Java Programm ist wie folgt aufgebaut:

```
public class name {
  public static void main(String[] args) {
    statement;
    statement;
    ...
  }
}
```

Jedes ausführbare Java Programm besteht aus einer Klasse `class`, welche eine Methode `main` enthält die verschiedene Anweisungen `statements` ausführt.

Kommentare helfen dabei, Programm-Codes verständlich zu machen. Sie können wie folgt implementiert werden:

```
// Entweder wird ein Kommentar mit einem doppelten Slash "//"

/*
 * Oder mit diesen Symbolen geschrieben
/*
```

2.2 Methoden

Eine **Methode** ist eine Sequenz von Anweisungen. Sie erlauben es, Wiederholungen zu vermeiden. *static methods* sind Methoden mit weiteren Eigenschaften. Als Beispiel die `main` Methode, welche beim kompilieren automatisch ausgeführt wird.

Der **control flow** bezeichnet die Abfolge der Ausführung von Anweisungen.

2.3 Typen und Variablen

Typen beschreiben Eigenschaften von Daten. Des weiteren bestimmt der Typ auch die Operationen welche auf die Daten ausgeführt werden können.

Primitive types sind in Java integrierte, einfache Typen. Dazu gehören:

- *int*: Ganze Zahlen
- *long*: Grosse ganze Zahlen
- *double*: Reelle Zahlen
- *char*: Einzelne Buchstaben
- *boolean*: Logische Werte (*true / false*)

Arithmetische Operatoren verknüpfen mehrere Werte/Ausdrücke. Beispiele dazu sind: +, -, x, /, %.

> ***Achtung:*** *Bei der Division oder dem Modulo mit einer int-Zahl ist das Ergebnis auch ein int! Beispiel: 14/4 = 3; 218%5 = 3;*

> ***Wichtig:*** *Die Umwandlung von Typen geschieht in Java automatisch! So wird zum Beispiel eine Division mit einem int und einem double zu einem double.*

Die *Precedence* bestimmt über die Reihenfolge wie Operatoren ausgeführt werden. Haben zwei Operatoren die selbe Assoziativität und Rangordnung, so werden diese *von links nach rechts ausgewertet*.

> ***Precedence in Java:*** *() vor x, /, % vor +, -.*

Eine **Variable** ist eine Name der es erlaubt auf gespeicherte Daten zurückzugreifen. Der Ablauf für das Brauchen von Variablen geht wie folgt:

1. *Deklaration:* Gibt den Namen und den Type der Variable an
   - `double myNumber;`
2. *Initialisierung:* Speichert den Wert der Variable
   - `myNumber = 12345;`
3. *Gebrauch:*Verwendung der Variable im Programm

Eine Variable kann nur Werte ihres eigenen Types speichern:

```
int x = 2.5; //ERROR: incompatible types
```

2.4 Loops

Loops erlauben wiederholte Ausführung einer (oder mehreren) Anweisungen. Ein *for-Loop* sieht wie folgt aus:

Syntax:

```
for (initialization; condition; update;) {
      statements;
      statements;
}
```

Ein Beispiel:

```
for (int i=0; i < Array.length; i++) {
      Array[i] = i;
}
```

2.5 Methoden mit Parametern

Syntax:

```
public static void methodName (type name) {
      statements;
}
```

Ein **Parameter** wird vom Aufrufer zur aufgerufenen Methode weitergegeben. Ein *formaler Parameter* ist einer, welcher in der Definition einer Methode deklariert wird. Der *actual parameter* ist der, welcher dann tatsächlich an die Methode weitergegeben wird.

Aufrufen einer Methode mit Parametern:

```
methodName(value1, value2, ...);
```

2.6 "if" - Anweisungen

if - Anweisungen werden nur ausgeführt, wenn (test) den Wert *true* zurück gibt. *if - else - Anweisungen* führen das if - Statement aus falls der test *true* zurück gibt, ansonsten das *else - Statement*.

Syntax:

```
if (test == true;) {
      statements;
} else {
      statements;
}
```

if- , for- und while-Loops verwenden alle **Boolsche Ausdrücke**. Diese Ausdrücke werden ausgewertet und ergeben entweder true oder false zurück. Zu den Boolschen Ausdrücken gehören:

- "==" gleich
- "!=" ungleich
- "<" kleiner
- ">" grösser

Boolsche Operatoren erlauben es, Vergleichsoperatoren zu verknüpfen:

- "&&" AND
- "| |" OR
- " ! " NOT

Precedence: *arithmetisch > vergleichs > boolsche*

2.7 Nochmals Schleifen

Kurzformen für einfache Zuweisungen erlauben es Statements der Form j = j+1; verkürzt darzustellen. Diese werden meistens in der Aktualisierung von Loop-Counters verwendet. Die in Java bekannten Kurzformen sind:

- x += 1;
- x -= 2;
- usw...

while - Schleifen führen den Loop solange aus wie der boolsche Wert *true* ergibt.

Syntax:

```
while (tests) {
      statements;
}
```

do-while - Schleifen sind eine leicht abgeänderte Version der while - Schleife. Der Unterschied liegt darin, dass der Test erst nach dem ersten Ausführen des Body's ausgewertet wird.

Syntax:

```
do {
      statement;
      ...
} while (test);
```

2.8 Ergebnis Rückgabe für Methode

Ein *Parameter* erlaubt die Kommunikation zwischen der aufrufenden zur aufgerufenen Methode. Ein *return value* erlaubt der aufgerufenen Methode dem Aufrufer einen Wert zu übermitteln.

Syntax:

```
public static type methodName (parameters) {
      statements;
      return expression;
}
```

Das *return statements* wertet die Expression aus und teilt diese der aufrufenden Methode mit.

2.9 Strings

Strings sind Abfolgen von Buchstaben. Sie werden initiiert durch:

```
String nameString = "Hello world!";
```

Strings erlauben den Zugriff auf einzelne Buchstaben. Ein String beginnt immer mit Index 0, die einzelnen Buchstaben sind des Types `char`. Des weiteren sind auch für Strings schon einige Methoden vordefiniert:

```
String.indexOf();
String.length();
String.substring(index0, index1);
String.equals(String2);
String.endsWith(String2);
...
```

> ***Wichtig:*** *Diese Methode kreieren stets einen neuen String und modifizieren nicht den bestehenden String!*

2.10 Input

Ein *interaktives Programm* liest einen Input aus der Konsole:

- *System.in:* Vordefiniertes Fenster für Input
- *System.out:* Vordefiniertes Fenster für Output

Scanner erlaubt es von verschiedenen Stellen (Konsole, WEB, Dateien, ...) Input zu lesen. Er ist in der *Java Bibliothek* java.util. definiert. Diese muss zu beginn des Programmes importiert werden, danach kann ein neues *Scanner-Objekt* konstruiert werden:

```
import java.util.*;

Scanner scannerName = new Scanner(System.in);
```

Der Scanner besitz bereits ein paar *vordefinierte Methoden*:

```
Scanner.nextInt();
Scanner.nextDouble();
Scanner.next();
Scanner.nextLine();
```

3. Arrays

Ein **Array** ist eine Reihe/Liste, die es uns erlaubt, mehrere Werte eines Types zu speichern und darauf zuzugreifen. Dabei beschreibt *Element* einen Eintrag im Array und *Index* den Ort des Elementes im Array (Index beginnt bei 0!).

Syntax:

```
type[] nameArray = new type[length];
```

Der *Zugriff* auf Elemente funktioniert mit:

```
nameArray[i] = value;
System.out.println(nameArray[i];)
nameArray.length;
```

4. Klassen und Objekte

4.1 Klassen und Objekte

Eine **Klasse** wird mit dem Keyword `class` deklariert, wobei die enthaltene Methode `main` immer automatisch aufgerufen wird beim Verwenden einer Klasse.

Ein **Objekt** ist der Sammelbegriff für alle Datenwerte, welche durch eine Klasse beschrieben werden (Bsp. ein Scanner, ein String, der Zufallsgenerator etc.). Ein Objekt muss *instanziert* werden, dies geschieht mit einem `new-Operator`:

Syntax:

```
Scanner console;
console = new Scanner(System.in);
Random rand = new Random();
```

Vergleichsoperatoren (wie <, ==, >=) funktionieren für Objekte nicht. Oft wird dafür eine `.equals()-Methode` verwendet. Beispielsweise bei Strings:

```
String name = console.next();
if (name.equals("Tom")) {...}
```

Folgende *String Methoden* sind uns bereits bekannt:

```
String.equals();        //Checks if two strings contain the same characters
String.equalsIgnoreCase(); //Same as above, ignores lower- and upper cases
String.startsWith();    //whether or not a String starts with a given char
String.endsWith();            //whether it contains a given char at the end
String.contains();            //whether a String contains a given char
```

Folgende *static Array Methoden* sind uns bereits bekannt:

```
Array.binarySearch(arr, val); //returns index of val in sorted Array
Array.copyOf(arr1, arr2);           //returns a new copy of an Array
Array.equals(arr1, arr2);           //true if same elements in same order
Array.fill(arr, val);               //fills the Array with a given value
Array.sort(arr);                    //sorts elements
Array.toString(arr);                //returns a String representing the Array
```

4.2 Array als Parameter

Will man einen *Array als Parameter* an eine Methode weitergegeben werden, dann funktioniert das wie folgt:

```
public static type methodName(type[] name) {...}
```

Das selbe gilt, wenn eine Methode einen Array als return-Value zurückgeben soll:

```
public static type[] myMethod (...) {
      ...
      return array;
}
```

Das Aufrufen dieser Methode sieht demnach wie folgt aus:

```
methodName(myArray);
```

4.3 Reference Semantics

Ein grosser Unterschied bei der Verwendung von Arrays im Vergleich zu Basistypen sind die **Reference Semantics**. Während beim Zugriff oder bei der Übergabe eines Basistyps der Wert der Variable weitergegeben wird, erlaubt ein Array den tatsächlichen Zugriff auf die gespeicherten Daten. Ein Array ist eine sogenannte *reference type variable*.

Dies hat für uns den grossen Vorteil, dass eine Methode einen Array als Parameter bekommen kann, ohne diesen kopieren zu müssen. Da ein Array wie oben erwähnt eine Referenzvariable ist, *speichert er keine Daten*, er verweist lediglich auf deren Speicherort.

Man unterscheidet demnach bei der Übergaben von Parametern zwischen **value semantics** (int, double,...) und **reference semantics** (String, Array,...).

4.4 Klassen selber entwickeln

4.4.1 Motivation

Um nochmals den Unterschied zwischen Klassen und Objekten zu zeigen:

- Eine Klasse beschreibt die Funktionalität von Objekten

- Objekte sind **Instances** einer Klasse

4.5 Attribute

Attribute werden dafür verwendet, den Zustand eines Objektes zu beschreiben. Attribute sind Variabeln in Objekten. Andere Klassen können auf die Attribute eines Objektes zugreifen:

```
variable.field;
variable.field = value;

Beispiel:
Point p1 = new Point();
System.out.println("the x-coordinate is: " + p1.x);
```

4.5.1 Random Klasse

Die *Java Random Klasse* liefert uns einen Zufallsgenerator (Wobei zu beachten ist, dass dies nur Pseudozufallszahlen sind). Folgende Methoden sind uns bereits bekannt in der Random Klasse:

```
Random.nextInt();       //returns a random integer
Random.nextInt(max);    //returns a random integer in range [0,max)
Random.nextDouble();    //returns a random double in. range [0.0, 1.0)
```

4.5.2 Math Klasse

Die *Java Math Klasse* liefert uns verschiedene mathematische Funktionen und Konstanten. Einige uns bereits bekannten Methoden sind:

```
Math.sqrt(doub);
Math.pow(a, b);
Math.Pi();
...
```

> ***Wichtig:*** *Teilweise liefern uns Math.-Methoden doubles, auch wenn wir ein int erwarten. Dies lässt sich mit einem* ***cast*** *umgehen. Beispiel:*

```
int x = (int) Math.pow(10, 3);
```

4.6 Methoden

Methoden beschreiben das *Verhalten eines Objektes*. Oft haben wir das Problem, dass wir Klassen entwickeln wollen, welche von den Klienten unabhängig sind.

Sogenannte *instance methods* oder auch *object methods* sind Methoden, welche innerhalb eines Objektes eine Klasse existieren und beschreibt sind.

Syntax:

```
public type name (parameters) {
      statements;
}
```

Achtung: *Der Syntax ist ohne static !!*

Beispiel:

```
public class Person {
      String name;
      int id;
      double[] hours;

      public double computePay {
            ...
      }
}

      Person p1 = new Person();
      p1.computePay;
```

Ein *implicit* Parameter beschreibt das Objekt, für welches eine Methode aufgerufen wird. Im obigen Beispiel ist *p1* in *p1.computePay* ein implicit Parameter.

Bis jetzt konnte jeder Klient auf alle Attribute eines Objektes zugreifen. Dies ist aber nicht immer erwünscht (im obigen Beispiel z.B.: Eine Person könnte den Stundenlohn ändern). Wir betrachten dazu zwei verschieden Arten von Methoden:

Accessor Methode: Eine Methode welche es erlaubt, den Zustand eines Objektes *anzusehen*. Z.B.: getAdress(), getOvertime(), ...

Mutator Methode: Eine Methode welche es erlaubt, den Zustand eines Objektes zu *verändern*. Z.B.: setTrainingTime(), setAge(), ...

Das Drucken von Objekten ist meistens nicht trivial. Es wird daher oft gefordert, eine *toString*-Methode zuschreiben, damit wird die Attribute von Objekten korrekt drucken können.

4.7 Konstruktoren

Konstruktoren dienen der Konstruktion und Initialisierung von Objekten. Ein *constructor* wird ausgeführt, sobald der new Operator ausgeführt wird und gibt keinen return-Statement.

Syntax:

```
public type(parameters) {
      statements;
}
```

> ***Wichtig:*** *Konstruktoren sind keine Methoden!*

Wird in einer Klasse keinen Konstruktor definiert, so ruft Java bei der Benützung des new Operators den *default constructor* auf. Dieser setzt alle Attribute auf ihre Standart-Werte (0, 0.0, null, false).

Beispiel:

```
public Person (String firstName, int age, int uniqueId) {
      name = firstName;
      agePerson = age;
      id = uniqueId;
}
```

Eine Klasse kann *mehrere constructors* haben, diese müssen sich aber in der Anzahl der Parameter unterscheiden!

4.8 Sichtbarkeit für Attribute

Encapsulation beschreibt die Arbeitsweise, Klassen als Mechanismus zur Abkapselung zu benutzen. Dabei werden Variablen ausserhalb einer Methode definiert und mittels Dot-Notaion darauf zugegriffen.

Möchten wir verhindern, dass von ausserhalb einer Klasse nicht auf ein Attribut zugegriffen werden kann, so können wir dieses mit dem Keyword *private* deklarieren. Syntax:

```
private type name;
```

> ***Achtung:*** *Der Zugriff auf Attribute in Methode der eigenen Klasse ist immer noch möglich!*

this ist ein Keyword, dass auf das Objekt verweist für welches eine Methode aufgerufen wird. Beispiel:

```
public Point (int x, int y) {
      this.x = x;
      this.y = y;
}
```

4.9 static Methoden und Variablen

Klassen könne auch als *Module* verwendet werden. Ein Beispiel dazu wäre Math, Arrays, System, usw. Ein Modul ist *wiederverwendbare Software*, die in einer Klasse abgelegt wird. Ein Modul hat **keine main-Methode**, die Klasse kann also nicht direkt ausgeführt werden.

Das Keyword **static** beschreibt Variablen und Methoden. Es beschreibt den Zustand, Teil einer Klasse, und nicht Teil eines Objektes, zu sein. Eine static Variable existiert nur einmal und Methoden aller Exemplare der Klasse können darauf zugreifen.

> ***Achtung:*** *static Variblen und Methoden sollte die Ausnahme sein!*

static Methoden sind Methoden einer Klasse. Sie können ohne Referenz auf ein Objekt aufgerufen werde.

Im Normalfall sind static Variablen private oder final (*final* bedeutet, dass der Wert nicht mehr geändert werden kann).

5. GUI und Daten In- und Output

5.1 Graphische Benutzeroberflächen (GUI)

GUI steht für *Graphical User Interface*. Der Vorteil eines GUI's ist, dass es vielseitiger als die Eingabe per Konsole ist. Ein GUI erlaubt die Eingabe via Maus, Gesten und ist oft *intuitiver und ansprechender*.

Die *Window-Klasse* erlaubt das Darstellen von einfachen Fenster ohne Steuerelemente oder anpassbares Layout. Sie wird wie folgt instanziert:

```
import gui.Window;
public class Empty {
      public static void main(String[] args) {
            Windwo window = new Window("Empty", 500, 300);
            window.open();
            window.waitUntilClosed();
      }
}
```

Die uns bereits bekannten *Window-Methoden* sind:

```
Window.open();                    //Window is opened
Window.close();                   //Window gets closed
Window.waitUntilClosed();     //Program waits until user closes window
Window.isOpen();              //Returns true if window is open
```

Weitere Methoden die wir kennen erlauben uns, Formen, Texte oder Bilder auf das Fenster zu zeichnen:

```
Window.setColor(R, G, B);                 //Sets the color to draw with
Window.fillRect(x, y, width, height);     //Draws a filled rectangle
Window.fillCircle(x, y, radius);          //Draws a filled circle
Window.fillOval();                  //Draws an oval
```

> ***Wichtig:*** *Das Koordinatensystem in Java beginnt oben links im Fenster bei (0,0).*

Im Moment sind alle unsere Formen die wir im Fenster zeichnen bestehend. Möchten wir aber den Inhalt des Fenster über Zeit änder, so brauchen wir weitere Methoden die es uns erlauben, den Fensterinhalt zu aktualisieren. Zu diesen Methoden gehören:

```
Windwo.refresh();                   //shows the current drawing
Window.refresh(int waitTime); //shows the current drawing after a
```

```
                                                    //certain time
Window.refreshAndClear();              //shows the current drawing but clears
                                                    //the screen befor that
```

Um das ganze noch spannender zu machen erlaubt uns die Window-Klasse auch verschiedene *Inputs* einzulesen. Dafür sind uns folgende Methoden bereits bekannt:

```
Window.isKeyPressed(String keyName);      //Checks if given key is presse
Window.isLeftButtonPressed();          //true if left mouse button is pressed
Window.isRightButtonPressed;           //true if right mouse button is pressed
Window.getMouseX();                    //returns the current x coordinate
Window.getMouseY();                    //returns the current y coordinate
```

5.2 File input

Die Klasse **File** erlaubt uns Operationen mit Dateien (Files) durchzuführen. Diese können wir mit dem folgenden Syntax instanzieren:

```
import java.io.*;
File file = new File("example.txt");
```

Folgende File-Methoden sind uns bereits bekannt:

```
File.exists();                         //true if file exists
File.canRead();                        //true if file can be read
File.getName();                        //returns the name of the file
File.length();                         //returns the size of the file in bytes
File.delete();                         //delets the file
File.renameTo(file);             //renames the file
```

> ***Wichtig:*** *Der Ausdruck new File("example.txt") erstellt keine Datei, sondern nur ein Objekt (Handler), welches für eine Datei mit diesem Namen steht (und welche eine Datei mit diesem Namen manipulieren kann).*

5.3 Exceptions

Eine **Exception** ist die Folge eines Fehlers zur Laufzeit des Programmes. Man sagt auch, dass das Programm einen Fehler wirft (*throws*). Dazu kann ein Programm auch Fehler auffangen (*catch*). Eine Methode kann spezifisch so deklariert werden, dass sie eine exception wirft, und diese nicht selber wieder auffängt:

```
public static void foo(...) throws type;
```

5.4 File output

Die **PrintStream**-Klasse erlaubt es, Daten auszugeben. Alle Methoden, welche wir aus der System.out-Klasse kennen, funktionieren auch in der PrintStream-Klasse (z. B. print(), println(), usw.). Syntax:

```
File file = new File("example.txt");
PrintStream ouptut = new PrintStream(file);

for (int i=0; i < 10; i++) {
      output.print("Hello Wolrd!")
}
```

5.5 Scanner im Einsatz

Ein *Token* ist eine folge von Zeichen, welche ein Scanner einliest. Mit den Methoden `.hasNextInt, .hasNextDouble` etc. können wir überprüfen, ob ein gesuchter Token vorliegt oder nicht. Beispiel:

```
if (console.hasNextInt()) {
      int age = console.nextInt();
} else {
      System.out.println("You didn't type in an integer");
}
```

Der Scanner *konsumiert* Tokens. Möchte man zum Beispiel einen Wert in einem File überspringen, so kann man dies mit `input.next();`machen.

Möchten wir ganze Zeilen eines Textfiles verarbeiten, so macht es Sinn, jede *Zeile einzeln* zu lesen und diese so zu verarbeiten. Dies kann mit `input.nextLine();` gemacht werden. Ein Scanner kann danach diese Zeile lesen:

```
String line = input.nextLine();
Scanner scanner = new Scanner(String);
scanner.next();
```

Das Java-Objekt **PrintStream** kann dafür verwendet werden, Daten auf Textfiles zu schreiben. Beispiel:

```
PrinStream printer = new PrintStream(new File("Beispiel"));
printern.println();
```

6. Arbeiten mit Objekten und Klassen

6.1 Einleitung

Manchmal möchten wir Daten verknüpfen (wie in einem Array), wissen aber nicht schon im Voraus, wie gross unser Input ist.

Dafür können wir die Klasse **ListNode** verwenden, sie kann einen Datenwert speichern und diesen mit einem nächsten Wert verlinken. Jeder Knoten dieser Liste speichert dabei einen Wert und einen Verweis zum nächsten Knoten. Beispiel:

```
ListNode list = new ListNode();
list.data = 42;
list.next = new ListNode();
list.data = 2;
list.next.next = new ListNode();
```

6.1.1 Beispiel LinkedList

Wir erstellen ein Beispiel für eine Klasse von verknüpften Listen (**LinkedList**).

Diese LinkedList besteht aus Nodes (Knoten), welche miteinander verknüpft sind.

```
public class ListNode {
	int data;
	ListNode next;

	public ListNode (int data) {
		this.data = data;
		this.next = null;
	}

	public ListNode (int data, ListNode next) {
		this.data = data;
		this.next = next;
	}
}
```

6.2 Entwurf von abgekapselten Klassen

Wir nehmen das Beispiel von Kapitel 6.1.1 wieder auf. Angenommen, wir wollen es nicht erlauben, dass ein Klient `ListNodes` direkt verändern kann. Wir kreieren also eine Klasse **LinkedIntList**, welche diese Knoten versteckt.

Diese Klasse soll sicher die Methoden `.add` und `.remove` haben. Die LinkedIntList besteht also aus eine Kette von Knoten. Die Klasse selber enthält ein Attribut *front*, welches auf den ersten Node zeigt.

```
public class LinkedIntList {
	private ListNode front;

	public LinkedIntList() {
		front = null;
	}
}
```

Wollen wir nun ein Node zur Liste **hinzufügen**, können wir dies zum mit einer `.add()` zum Beispiel so lösen:

```
public void add(int value) {
	if (front == null) {
		front = new ListNode(value);
	} else {
		ListNode current = front;
		while (current.next != null) {
			current = current.next;
		}
		current.next = new ListNode(value);
	}
}
```

Wollen wir eine einen Node **bekommen**, können wir die `.get()` Methode so implementieren:

```
public int get(int index) {
	ListNode current = front;
	for (int i = 0; i < index; i++) {
		current = current.next;
	}
	return current.data;
}
```

Wollen wir ein Node **entfernen**, können wir das mit einer `.remove()` Methode wie folgt tun:

```
public void remove() {
	if (index == 0) {
		front = front.next;
	} else if (front == null {
		System.exit(-1);
	} else {
		ListNode current = front;
		for (int i = 0; i < index -1; i++) {
			current.next;
		}
		curent.next = current.next.next;
```

```
        }
}
```

6.2.1 Variationen vom Beispiel LinkedIntList

Häufig brauchen wir eine *Datenstruktur d*, sodass wir:

- Am Anfang einen int Wert `value` einfügen können
    - d.add(0, value) —> d.push(value)
- Den ersten Wert entfernen können
    - int result = d.remove() —> d.pop()
- Prüfen, ob es überhaupt Werte in der Datenstruktur gibt
    - d.isEmpty()

Diese Datenstruktur ist auch als *Stack* bekannt. Beispiel:

```
public class IntStack {
      private ListNode front;
      public IntStack() {
            front = null;
      }

      public void push (int value) {...}
      public int pop() {...}
      public boolean isEmpty() {...}
}
```

Noch ein weiteres Beispiel: Wir wollen ein Node in eine Liste in geordneter Reihenfolge einfügen:

```
public void addSorted (int value) {
      if (front == null || value <= front.data) {
            front = new ListNode(value, front);
      } else {
            ListNode current = front;
            while (current.next != null && current.next.data < value) {
                  current = current.next;
            }
      current.next = new ListNode(value, current.next);
      }
}
```

6.3 Hinweise und Regeln für verständliche Programme

Hinweise für **Verständlichkeit**:

- Einfache Regeln
- Möglichst klar
- Möglichst kompakt (No repeat)
- Nicht mehr als 100 Zeichen pro Zeile
- Reihenfolge in Datei
    - Import
    - (Haupt) Klasse
    - main Methode
    - andere Methoden

Regeln für **Namen**:

- Nur Buchstaben und Ziffern
- Klassennamen; MitGrossBuchstaben
- Methodennamen: Wenn möglich mit verb, erster Buchstabe klein
- Variablenamen: beschreibend

Regeln für **Variablenamen**:

- Kurze Variablenamen sind reserviert für Loopcounter (Bsp. “i”)
- Kein Typ und Metainformationen im Namen
- Masseinheit soll wenn relevant Teil des Namens sein

6.3.1 Code-Regeln

Faktorisierung (factoring) beschriebt das Herausarbeiten von gemeinsamen/redundanten Anweisungen. Beispiel:

Der Code-Block:

```
if (a == 1) {
      System.out.println(a);
      x = 3;
      b = b + x;
} else if (a == 2) {
      System.out.println(a);
      x = 6;
      y = y + 10;
      b = b + x;
} else { // a == 3
      System.out.println(a);
      x = 9;
      b = b + x:
}
```

Ist äquivalent zum Block:

```
System.out.println(a);
x = 3*a;
if (a == 2) {
      y = y + 10;
}
b = b + x;
```

6.4 Objektexemplare in Programmen

Die Klasse *Rational* dient zum Darstellen von rationalen Zahlen. Doubles sind nur Approximationen! Zu den Attributen der Rationalen zahlen gehören `num`, der Zähler, sowie `den`, der Nenner. Beide Attribute sollten private sein.

6.4.1 Speicher und Adressen

Für eine deklarierte Variable muss das Laufzeitsystem Speicherplatz finden. Der Speicher des Computers ist dabei in verschiedene Bereiche unterteilt, wobei es einen spezifischen Teil gibt, welcher für unser Java Programm reserviert ist.

Objekte die durch den new Operator erschaffen werden, werden im *heap* (Halde) abgelegt. Für jede aufgerufene Methode stellt das Laufzeitsystem einen neuen Bereich zur verfügung, welcher *Stack Frame* genannt wird. All diese Stack Frames werden in einem *Stack* organisiert.

6.5 Mehr Optionen für Sichtbarkeit

6.5.1 Java Packages

Ein **Package** ist eine Ansammlung von Klassen die zusammengehören. Eine Datei kann dabei nur zu einem Package gehören. Die Deklaration einer Package kann wie folgt erfolgen:

```
package packageName.subPackage;

public class Name {...}
```

In diesem Beispiel sollte die Datei `Name.java` im Folder `packageName/subPackage`sein.

Das **default Package** ist das Package, mit der wir bis jetzt gearbeitet haben. Dateien (Klassen), welche keine Package-Deklaration enthalten, gehören der namenlosen default Package an. Diese können *nicht importiert* werden.

6.5.2 Packages und Sichtbarkeit

In Java gibt es folgende **access modifiers**:

- *public* —> Sichtbar für alle anderen Klassen (nach Import)
- *private* —> Sichtbar nur in dieser Klasse
- *protected* —> Sichtbar nur in dieser Klasse, allen Unterklassen und allen anderen Klassen der Package

7. Vererbung, Abstraktion und Polymorphismus

7.2 Neue Klassen aus bestehenden kreieren

Vererbung (**inheritance**) erlaubt uns, eine Klasse als Erweiterung einer anderen Klasse auszudrücken. Dies ist eine sogenannte `is-a` Beziehung. Beispiel: *Ein Arzt IST EIN Angestellter*. Die **inheritance hierarchy** beschreibt eine Menge von Klassen, verbunden durch eine is-a Beziehung, die gemeinsamen Code verwenden.

Eine Klasse kann eine andere erweitern (*extend*) und Daten, Zustand sowie Verhalten absorbieren. Syntax:

```
public class name extends superclass {...}
```

Eine **subclass**, also eine Klasse die von einer **superclass** abstammt, erbt alle Methoden und Attribute von dieser! *Override* beschreibt den Vorgang, eine in einer superclass definierten Methode in einer subclass neu zu definieren. Dazu ist kein speziellen Syntax notwendig, es reicht einfach die Klasse neu zu schreiben.

Subclasses können dabei eine Methode der superclass aufrufen, auch wenn diese neu definiert wurde. Syntax:

```
super.methodName();
```

7.3 Vererbung und Konstruktoren

Konstruktoren werden *nicht geerbt*! Wir können den Konstruktor der superclass in einer subclass aber spezifisch aufrufen. Syntax:

```
public class SubClass (int value) {
      super(value);           //calls superclass constructor
}
```

Wichtig: Auf private Attribute kann in der subclass nicht direkt zugegriffen werden!

Diese Problem wird mit getter- und setter-Methoden behoben.

Definieren wir in der superclass einen *Konstruktor mit Parametern* erstellt, so müssen in den subclasses auch neue Konstruktoren erstellt werden! In diesem fall kann der *default constructor super();* nicht mehr aufgerufen werden!

7.4 Selektives Verhalten von von Objekten festlegen

Wollen wir in subclasses auf Attribute zugreifen, auf welchen in anderen subclasses vielleicht nicht zugegriffen werden soll, können wir das Keyword **protected** benutzen. Es erlaubt den Zugriff *innerhalb der Klasse und ihrer subclasses*.

Die `ist-ein` Verbindung beschreibt einen Zusammenhang zwischen der superclass und ihrer subclasses. Jede Instanz vom Typ einer subclass, ist automatisch auch vom Typ der superclass. Instanzen einer *superclass können also auf eine subclass verwiesen* werden, umgekehrt gilt dies nicht.

7.5 Die Klasse Object

Die Klasse **Object** ist die Urahn aller Klassen. Sie hat keine Superclass und alle Klassen sind *subclasses von Object*. Eine Referenzvariable für Object kann auf Exemplare jeder Klasse verweisen!

Der **instanceOf** Operator erlaubt zu prüfen, ob eine Variable auf ein Object eines gewünschten Typen verweist. Beispiel:¨

```
String s = "hello";
Point p = new Point();

if(s instanceof Point){
      return true;
}
```

Beispiel einer `equals-Methode` :

```
public boolean equals(Object o) {
      if(other instanceof Point) {
            Point other = (Point)o;
            return x == other.x && y == other.y;
      } else {
            return false;
}
```

Wichtig: *Die* ***cast-Umwandlung*** *ändert eine Object Referenz in eine gewünschte andere Referenz. Dies ändert nicht die Referenz. Eine cast-Umwandlung funktioniert in der Inheritance Hierarchie nur* ***nach oben und nach unten****, nicht seitlich!*

7.6 Polymorphismus

Mit **Polymorphismus** können wir Programme erstellen, die mit Exemplaren verschiedener Klassen arbeiten, wobei die Klassen in einer Vererbungshierarchie angeordnet sein müssen.

Der *Typ der Referenzvariable* bestimmt die Methoden die für Objekte aufgerufen werden können, wobei der aktuelle Subtyp des Exemplares die Version der aufgerufenen Methode bestimmt.

8. Interfaces

8.1 Interfaces in Java

Ein **Interface** ist eine *Schnittstelle* die es erlaubt, eine Menge von Methoden vorzuschreiben, die von einer Klasse implementiert werden müssen. Beispiel:

```
public interface Shape {
      public double area();
      public double perimeter();
}
```

Interfaces enthalten also *abstrakte Methoden*, also Methoden die nur aus einem Header bestehen und keinen Body haben. Die Implementierung eines Interfaces sieht wie folgt aus:

```
public class className implements interface {
      ...
}
```

Die genauen Details einer abstrakten Methode werden also von einer Klasse definiert, die *Parameter*, der *Return* wird aber vom *Interface* bestimmt.

Auch Interfaces können erweitert werden! Syntax:

```
public interface name extends name1, name2 {
      ...
}
```

9. Exceptions

9.1 Übersicht

Eine **Exception** beschreibt eine Änderung der Ausführungsreihenfolge auf Grund eines ausserge-wöhnlichen Ereignisses. Wir unterscheiden grob zwischen zwei Arten:

- Exceptions, die vom Programm behandelt werden könne. Beispiel: *FileNotFound-Exception*

- Exceptions, die nicht vom Programm behandelt werden können. Beispiel: *OutOf-MemoryError* oder *NullPointerException*

Wichtig: *Exceptions, die vom Programm behandelt werden können, müssen vom Programm behandelt werden! Dies funktioniert entweder durch Ankündigen oder durch einen try-catch Block.*

Beispiel *throws* (Ankündigen):

```
public static Scanner fileReader (Scanner input) throws FileNotFoundException {
      String name = input.next();
      return new Scanner(new FileReader(name));
}
```

Beispiel *try-catch* Block:

```
public static Scanner fileReader (Scanner input) {
      Scanner reader = null;

      while (rd == null) {
            try {
                  String name = input.next();
                  rd = getScanner(name);
            } catch (FileNotFoundException e) {
                  System.out.println("Can't open file");
            }
      }
      return rd;
}
```

10. Generische Programmierung

10.1 Einleitung

Das Ziel der generischen Programmierung ist es, *möglichst vielseitig verwendbare Software* zu schreiben. Ein Interface zum Beispiel erlaubt es, gesetzte Methoden an vielen verschiedenen Orten zu benützen, jedoch ist auch in den Methoden im Interface der Typ von return-Values und Parametern bereits festgelegt.

Eine **Collection** ist ein Objekt (bzw. eine Sammlung) das Daten speichert. Eine Collection besitz verschiedene Methoden, zu denen gehören typischerweise *add, remove, clear, size* usw.

10.2 ArrayList

ArrayList ist eine von Java zur Verfügung gestellte Klasse (ein Listenobjekt). Im Unterschied zum `array[]` stellt ArrayList eine dynamische Liste zur verfügung (Grösse kann also geändert werden).

Folgende `ArrayList-Methoden` sind uns bekannt:

```
ArrayList.add(value);               //Fügt Wert am Ende der Liste an
ArrayList.add(index, value);  //Fügt Wert an bestimmtem Index hinzu
ArrayList.clear();                        //Leert die Liste
ArrayList.indexOf(value);           //Gibt den  Index eines Wertes zurück
ArrayList.set(index, value);  //Ersetzt den Wert an gegebenen Index
ArrayList.toString();               //Gibt den Array in String-Form zurück
ArrayList.size();                   //Gibt die Länge der Liste zurück
ArrayList.remove(value);            //Entfernt den gebenen Wert aus dem Array
```

In einem Array können wir ein beliebiges Objekt abspeichern. Dies erfolgt so:

```
ArrayList<Type> name = new ArrayList<Type>();
```

Dieser *Type-Parameter* kann auch bei der Definition einer eigenen Klasse gebraucht werden. Beispiel:

```
class MyType<T> {
      T intern;

      public myType(T i) {
            intern = i;
      }
```

```
}
```

Man benutzt normalerweise einen einzelnen Grossbuchstaben als Namen für einen Typenparameter. Folgende Konvention gilt:

- E - Element
- K - Key
- N - Number
- T - Type

Wichtig: *In einer ArrayList können nur Referenzvariablen gespeichert werden!*

Möchte man in einer ArrayList einen Basistyp speichern, gibt es dazu sogenannte **Wrapper Klassen**. Aus `int` wird als `Integer`, aus `double` wird `Double` etc.

10.3 Vergleichen von Objekten

Normalerweise sollte in einer Klasse eine **compareTo - Methode** definiert werden. Diese sollte eine totale Ordnungsrelation definieren, also A < B, A > B und A = B.

Diese Methode sollte bei einem Vergleich eine Zahl zurückgeben, die folgender Konvention entspricht:

- eine Zahl > 0, wenn `other` nachher in der Ordnung kommt
- eine Zahl < 0, wenn `other` vorher in der Ordnung kommt
- 0 wenn `other` die gleiche Ordnung hat

Wollen wir also zum Beispiel eine ArrayList mit einem eigenen Typ sortieren (mit `Collections.sort(ArrayList)`, dann muss die Klasse dieses Types eine compareTo-Methode definieren. Dazu muss unsere Klasse das Interface **Comparable** implementieren.

Syntax:

```
public class Name implements Comparable<Name> {
      ...
      public int compareTo (Name other) {
```

```
            if(this ">" other) {
                  return 1;
            } else if(this "<" other) {
                  return -1;
            } else {
                  return 0;
            }
      }
}
```

10.4 Mengen

Ein **Set** (Menge) ist eine Ansammlung von eindeutiger und einzigartiger Elemente (keine Duplikate!) für welche folgende Operationen ausgeführt werden können:

- add
- remove
- contains

Ein Set wird durch verschiedene Klassen implementiert:

- *HashSet*: Basiert auf einem HashTable, erlaubt O(1) für alle Operationen.
- *TreeSet*: Implementierung basiert auf einem binären Baum, erlaubt O(log(n)) für alle Operationen. -> Benötigt `comapreTo-Methode`
- *LinkedHashSet*: Speichert die Elemente in der Reihenfolge, in der sie hinzugefügt wurden, O(1) für alle Operationen.

Für ein Set sind uns folgende Methoden bekannt:

```
Set.add(value);              //Fügt den gegebenen Wert zum Set hinzu
Set.contains(value);    //Gibt true zurück wenn der Wert im Set ist
Set.remove(value);           //Entfernt den Wert vom Set
Set.clear();                 //Entfernt alle Elemente im Set
Set.isEmpty();               //Gibt true zurück wenn das Set leer ist
Set.toString();              //Gibt das Set in String-Form zurück
```

Um über alle Elemente einer Ansammlung (`List` oder auch `Set`) zu iterieren gibt es eine *for each* Schleife. Diese sieht wie folgt aus:

```
for (Type variable : collection) {
```

```
        statements;
}
```

10.5 Abbildungen

Eine **Map** (Abbildung) enthält eine Menge von *Keys* (Schlüsseln) und eine Ansammlung von *Values* (Werten), wobei jeder Schlüssel einem Wert assoziiert ist.

Folgende Implementationen einer Map sind uns bekannt:

- *HashMap*: Schlüssel sind in einem Array gespeichert, O(1) für alle Operationen.
- *LinkedHashMap*: Speichert die Elemente in der Reihenfolge, in der sie hinzugefügt wurden, O(1) für alle Operationen.
- *TreeMap*: Schlüssel werden in einem binären Suchbaum gespeichert, O(log(n)) für alle Operationen.

Syntax:

```
Map<KeyType, ValueType> name = new MapType<KeyType, ValueType>();
```

Folgende Map-Methoden sind uns bekannt:

```
Map.put(key, value);    //Fügt Schlüssel mit Wert hinzu
Map.get(key);                 //Gibt den Wert assoziiert zum Schlüssel zurück
Map.containsKey(key);   //Gibt true zurück wenn es den Schlüssel gibt
Map.remove(key);        //Entfernt das Schlüssel-Wert Paar
Map.clear();                  //Leert die Map
Map.size();                   //Gibt die Anzahl von Key-Value Pairs zurück
...
```

Eine wichtige Methode ist das `.keySet()`. Diese Methode liefert die Menge aller Keys, welche in einer Map enthalten sind, zurück. Analog dazu liefert die `.values()`-Methode aller in einer Map enthaltenen Werte als Set zurück.

10.6 Iteratoren

Ein Problem bei Iterieren über ein Set/Map ist, dass die `for each` Loop nur einen *read-only* Zugriff auf die Menge. Das heisst also, dass die Menge während dem Iterieren *nicht bearbeitet* werden darf.

Ein **Iterator** ist ein Objekt das dem Klienten erlaubt, die Elemente einer Ansammlung zu besuchen und diese zu entfernen.

Syntax:

```
Set<Type> nameSet = new HashSet<Type>();
Iterator<Type> itrName = nameSet.iterator();
```

Folgende Iterator-Methoden sind uns bekannt:

```
Iterator.hasNext();           //Gibt true zurück wenn es noch weitere
                                    //Elemente gibt
Iterator.next();        //Gibt das nächste Element im Set zurück
Iterator.remove();            //Entfernt das letzte Element von .next()
```

11. Systematische Programmierung

11.2 Aussagen

Eine **Assertion** (Aussage) ist eine Behauptung die entweder wahr oder falsch ist.

Wir setzten verschieden Punkte in unserem Programm fest, an welchen wir uns eine Logische Aussage (Frage) stellen. Dabei ist zu beachten, dass der ganze Code vor dem Punkt ausgeführt wurde und der Code danach noch nicht. Eine Aussage können wir danach bestimmen mit:

- NIE
- MANCHMAL
- IMMER

11.3 Aussagen für Anweisungen

13.3.1 Einleitung

Die **Hoare Logik** ist ein Ansatz wie man über ein Programm logische Schlüsse ziehen kann. Sie basiert im Wesentlichen aus zwei Schritten:

1. Vorwärts und rückwärts schliessen
2. Genauere Definition von Aussagen, Vor- und Nachbedingungen

Dieses Vorgehen ist eine gute Schulung, systematisch zu programmieren.

- Wir können über Zustände eines Programmes schlussfolgern
- Wir können den Effekt eines Programmes beschreiben
- Wir können eine Aussage als “weaker” und “stronger” beschreiben

Beispiel zum *vorwärts Schliessen:*

```
// Annahme: w > 0
```

```
x = 17;
// w > 0 AND x == 17
y = 42;
// w > 0 AND x == 17 AND y == 42
z = w + x + y;
// w > 0 AND x == 17 AND y == 42 AND z > 59
```

Beispiel zum *rückwärts Schliessen:*

```
// w + 17 + 42 > 0 (w > -59)
x = 17;
// x + w + 42 > 0
y = 42;
// x + y + w > 0
z = w + x + y;
// Gewünscht x > 0
```

Die **Precondition** ist eine Annahme, die vor der Ausführung eines Statements gilt. Die **Postcondition** ist eine Aussage, die nach der Ausführung eines Statements gilt. Pre- und Postconditions sind Boolsche Ausdrücke die sich auf den Zustand eines Programmes beziehen.

Die Grundidee zum Schliessen über if-else-Statements sieht wie folgt aus:

1. Die Precondition für den if-Block und den else-Block beinhaltet das Ergebnis des Tests.

2. Die Postcondition nach dem if-Statement ist die Disjunktion der Postconditions des if- und else-Blockes.

Beispiel:

```
// Annahme: x >= 0
z = 0;
// x >= 0 AND z == 0
if (x != 0) {
      // x >= 0 AND z == 0 AND x != 0 (also x > 0)
      z = x;
      // x >= 0 AND z == 0 AND x != 0 AND z > 0
      //(also x > 0 AND z > 0)
} else {
       // x >= 0 AND z == 0 AND !(x != 0) (also x == 0)
      z = x + 1;
      // x >= 0 AND z == 0 AND !(x != 0)
      //(also x == 0 AND z == 1)
}
// (x > 0 AND z > 0) OR (x == 0 AND z == 1) (also z > 0)
```

Üblicherweise werden Pre- und Postconditions in `{ }`-Blöcken geschrieben (*kein Java mehr!*).

11.3.2 Hoare Tripel

Ein **Hoare Tripel** besteht aus:

- `{ P }`: Precondition
- `S`: Programmsegment
- `{ Q }`: Postcondition

Ein Hoare Tripel ist entweder *gültig* oder *ungültig*:

- Gültig: Für jeden Zustand, für den P gültig ist, ergibt die Ausführung von S immer einen Zustand, für welchen Q gültig ist.
- Ungültig: Andernfalls.

Beispiele:

```
{x != 0} y = x*x; {y > 0}                    /Gültig
{z != 1} y = z*z; {y != z}                   /Ungültig
{true} (x = y; z = x;) {y == z}              /Gültig
```

11.3.3 Schwächste Vorbedingung

Nehmen wir an wir haben zwei Aussagen P1 und P2. Wenn es nun der Fall ist, dass P1 => P2 (also P1 impliziert P2) gilt, dann sagen wir, dass P1 *stronger* und P2 *weaker* ist.

Beispiele:

- `x = 17` ist stärker als `x > 0`
- `x is prime` ist weder stärker noch schwächer als `x is odd`

Ohne Schleifen und Methoden gibt es für jedes Programmsegment S und jede Postcondition eine **eindeutige schwächste Precondition**, abgekürzt `wp(S,Q)`.

Beispiel:

```
wp(x = y*y, x > 4) ist (y*y > 4), das heisst |y| > 2
```

11.4 Preconditions und Postconditions für Loops

Um Aussagen über die Ausführung von Loops zu machen brauchen wir eine **Invariante**. Zudem muss die Precondition für die Schleife die Invariante implizieren (D.h. die Precondition muss stärker oder gleich stark wie die Invariante sein).

In Hoare-Logik wäre das also:

- `{ P } while(B) S; { Q }` wobei P die Precondition, B der Schleifentest, S der Schleifenrumpf und Q die Postcondition ist.

Wir müssen also eine Invariante finden, sodass gilt:

- `P => I`: Die Invariante gilt zu beginn
- `{I AND B} S {I}`: Nach ausführen des Rumpfes gilt die Invariante wieder
- `(I AND !B) => Q`: Invariante und das Verlassen des Loops impliziert die Postcondition

Beispiel:

```
{x >= 0}
y = 0;
i = 0;
{pre: x >= 0 AND y == 0 AND i == 0}
{inv: y == sum(1, i)}
while(i != x) {
      i = i+1;
      y = y+1;
}
{post: i == x AND y == sum(1,i)}    (also y == sum(1,x))
```

11.5 Objekt Invarianten

In einer Situation wo alle Exemplare eine gewisse Bedingung erfüllen müssen können wir **Objekt Invarianten** definieren. Diese müssen vor und nach der Ausführung aller Methoden gelten (jedoch nicht unbedingt während dem Ausführen).

Beispiel:

Wir wolle in einem Objekt eine Zeit in Stunden und Minuten festhalten:

```
public class ZeitSpanne {
      int stunden;
      int minuten;
}
```

Für diese Klasse können wir nun die Invariante stellen, dass *das Attribut* `minuten` *zwischen 0 und 59 (einschliesslich) liegen muss.* Unser Konstruktor könnte dan so aussehen:

```
public ZeitSpanne (int stunden, int minuten) {
      if (stunden < 0 || minuten < 0) {
            throw new IllegalArgumentException();
      }
      this.stunden = stunden + minuten/60;
      this.minuten = minuten % 60;
}
```

Diese Invariante führt aber dazu, dass wir in jeder Methode wo wir die Werte von Stunden und/oder Minuten verändern, die Invariante wiederherstellen müssen.

```
this.stunden += this.minuten/60:
this.minuten = this.minuten % 60;
```

11.6 Entwurf von Klassen

Verschiedene Heuristiken zum Entwurf von Klassen zusammengefasst:

- Daten sollten innerhalb einer Klasse geschützt sein (vor Modifikation und Zugriff)
- Eine Klasse soll nur eine Abstraktion erfassen
- Klassen sollten klein sein
- Vererbungshierarchien sollten tief sein - je tiefer desto besser (aber nicht tiefer als 5-7 Ebenen)
- Vererbungen sollten früh in der Entwicklung berücksichtig werden

Die **Heuristik für den Entwurf** einer Klasse lässt sich also in vier zusammengefasste Schritte aufteilen:

1. Klassen identifizieren

2. Beziehungen zwischen Klassen erarbeiten

3. Attribute festlegen

4. Methoden festlegen

Oft hilft es in einer Satz-Aufgabe die Nomen herauszuschreiben, aus diesen können oft die Klassen herausgearbeitet werden!

12. Abstrakte Daten Typen

12.1 Polymorphismus und ADT's

12.1.1 Einführung

Eine Klasse soll grundsätzlich eine *minimale Schnittstelle* definieren, die von allen anderen Klassen genutzt werden kann.

Ein **Abstrakter Daten Typ**, kurz **ADT**, ist eine Spezifikation einer Ansammlung von Daten und Operationen, die mit diesen Daten ausgeführt werden können. In Java werden ADT's mit Interfaces beschrieben, zum Beispiel: `Collection, Deque, List, Map, Set,...`

Es ist grundsätzlich eine gute Idee die Variablen für Ansammlungen als *Variable des ADT* Interface Typs zu deklarieren. Beispiel:

```
List<String> list = new Arraylist<String>();
```

Das selbe gilt auch für die Deklaration von Parametern:

```
public void methodName (List<String> string, ...) {...}
```

12.1.2 Stack

Ein Beispiel für einen ADT wäre das **Stack**. Ein Stack ist grob gesehen eine Ansammlung zu welcher Elemente hinzugefügt und entfernt werden können. Ein Stack ist *LIFO* (Last-In, First-Out), ein Element wird also in der umgekehrt Reihenfolge entfernt, wie es hinzugefügt wurde.

Folgende Operationen sind in einem Stack wichtig:

```
Stack.push(value);          //Legt den gegebenen Wert oben aufs Stack
Stack.pop();                //Entfernt und gibt das oberste Element zurück
Stack.peek();               //Gibt das oberste Element zurück
```

12.2 Abstrakte Klassen

Eine **abstrtact class** (Abstrakte Klasse) ist eine Mischform zwischen einem Interface und einer Klasse. Sie tut folgendes:

- Definiert einen Superclass-Typ mit Methodendeklaration (wie Interface) und vollständigen Methoden (wie Klasse)

- Definiert Attribute (wie Klasse) und/oder Konstanten (wie Interface)

Achtung: *Abstrakte Klassen können nicht instanziert werden!*

Syntax:

```
public abstract class nameClass {
      ...
      //abstract method
      public abstract type nameClass(parameters);
}
```

Eine **abstract method** muss mit dem Keyword `abstract` deklariert werden und Parameter, Rückgabetyp und Sichtbarkeit festlegen.