Einführung in die Programmierung

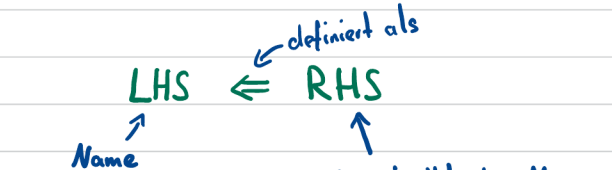
EBNF

Extended Backus Naur Form

Wie kann man eine Programmiersprache beschreiben?
→ EBNF beschreibt die Syntax mit 4 Elementen und ist formal & präzise

Eine EBNF Beschreibung besteht aus einer Menge an EBNF Regeln.

Aufbau einer Regel:



Beispiel:

<digit7> ⇐ 7

Die RHS. kann einen Namen einer anderen EBNF Regel, einen Buchstaben/Zeichen oder eine Kombination der vier Kontrolelemente enthalten.

Die Kontrolelemente

1. Aufreihung / sequence
2. Entscheidung / decision mit Auswahl oder Optionen
3. Wiederholung / repetition
4. Rekursion / recursion

Aufreihung

- von links nach rechts
- Reihenfolge ist wichtig

<digit_7> ⇐ 7
<digit_77> ⇐ 77

Auswahl

- eine Menge von Alternativen
- getrennt durch |
- Reihenfolge unwichtig

<bsp> ⇐ Beispiel | Bsp.

Klammern für Klarheit

Optionen

- Elemente in [ ]
- kann gewählt werden, muss aber nicht

<vorzeichen> ⇐ [+ | -]

ε epsilon
wenn nichts

Wiederholung

- Element in { }
- kann 0, 1, 2, ... mal wiederholt werden

<folge> ⇐ {<digit>}

Rekursion

- eine Regel ist rekursiv, wenn die RHS den Namen der Regel selbst enthält

für Klarheit

r ⇐ ε | (A <r>)

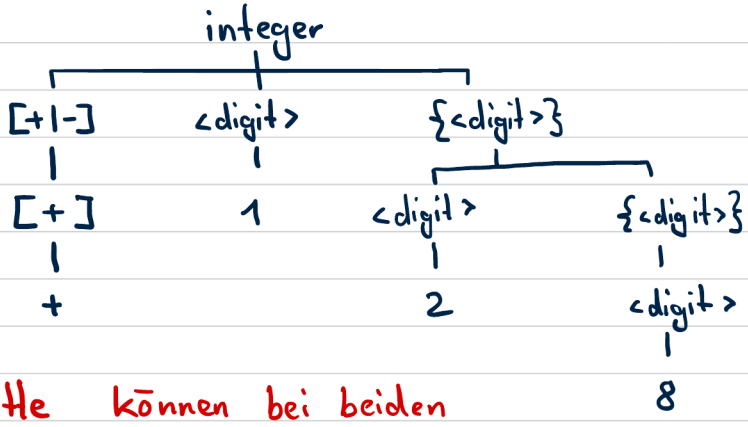
Es gibt verschiedene Möglichkeiten zu zeigen, dass ein Symbol für eine gegebene EBNF Beschreibung gültig ist.

Tabelle

integer				
[+|-]	<digit>	{<digit>}	
[+]	"	"	
+	"	"	
+	1	"	
+	1	<digit>	{<digit>}
+	1	2	"
+	1	2	8
+	1	2	8

Anfang
LHS durch RHS ersetzen
1. Auswahl gewählt
Option gewählt
<digit> ersetzt und auswahl
1. Wiederholt
<digit> ersetzt und auswahl
nochmals wiederholt, ersetzt und ausgewählt

Ableitungsbaum

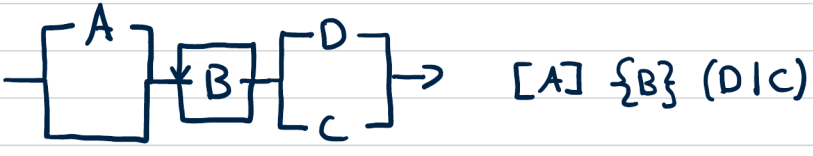


Triviale Schritte können bei beiden Optionen kombiniert werden.

Damit wir auch Sonderzeichen wie { in einer Regel verwenden können, schreiben wir sie in einem Kasten [{]

Zwei EBNF Beschreibungen sind nur dann äquivalent, wenn jedes mögliche Symbol von beiden als legal (oder illegal) erkannt wird.

Eine EBNF Regel kann auch als Graph dargestellt werden.



Dabei stellt ein Pfad durch den Graphen ein legales Symbol dar.

Java

Dokumentation ist essenziell für gute lesbarkeit von Programmen. Dazu benutzen wir Kommentare. Kommentare sollten an folgenden Orten stehen:

- Anfang eines Programmes
  - Autor
  - Algorithmus
  - Zweck
- Anfang jeder Methode
- Vor Code der nicht leicht verständlich ist.

Zur besseren Dokumentation, geben wir immer die Typen an.

Typen:
- fest eingebaute („primitive types")
- aus standard Bibliotheken (Bsp. string)
- eigene Typen

8 primitive types:
- boolean
- byte
- char
- int
- short 16 byte int
- long 64 byte int
- double 64 byte
- float 32 byte

! double is stronger than int, e.g.
2/2.0 = 1.0 3+2.0 = 5.0
result is double (automatic type casting) !

Literals sind Werte, welche direkt im Programm erscheinen. (hard-coded)

Operatoren sind entweder links- oder rechts-assoziativ. Dazu kommt noch ein Rang um die Ordnung zwischen verschiedenen Operatoren zu entscheiden. Wenn Rang und Assozialivität gleich sind wird von links nach recht gelesen. Klammern helfen dies zu vereinfachen.

Type casting wird mit (type) gemacht. Bsp. (double) 2
Dies kann aber auch implizit gemacht werden. Bsp. 1.0 / 2

Variable ist ein Name, der uns erlaubt auf einen gespeicherten Wert zuzugreifen.

for (initilization, test, counter) {
}
kann auch schon vorhanden sein

Funktionen / Methoden

public static void funcName (type name) {return}
return type
parameter

funcName(1)
argument

Value semantics dass kopieren eines Wertes

Boolsche Operatoren

&&	and
||	or
!	not

haben tiefere Präzedenz

Short-Circuit auswertung wird abgebrochen wenn Auswertung fest steht. Z.Bsp. LHS von && false ist.
Dies ist nicht der Fall wenn | oder & verwendet wird.

Bei i++ gilt zuerst auswerten, dann modifizieren.

Das Scope ist die Sichtbarkeit von Variablen.
Es gilt das Variablen zuerst deklariert werden müssen.
Danach sind sie bis zum Ende ihres Blockes ({...}) verfügbar.

Arrays

`type[] name = {1,2,3};` initialisierung mit literals

`type[] name = new type[length];` initialisierung mit defaults.

defaults: int: 0 boolean: false
double: 0.0 string: null

Wichtige Funktionen: name.length

von Arrays Klasse → Arrays.equals(a_1, a_2)
- sort
- toString

ArrayList

Verwendet ein dynamisches Array um die Werte zu speichern, manipulieren von einträgen ist langsam aber Zugriff und speichern ist schnell.

LinkedList

Verwendet eine doubly linked list und ist daher schnell im manipulieren von Daten aber langsamer beim Zugriff und braucht mehr Speicher.

Sets

Sets sind Mengen, wobei ein Element nur einmal vorkommen darf. Es braucht dafür eine Ordnungsrelation.

TreeSet O(log n)

- Elemente werden in Reihenfolge gespeichert.

HashSet O(1)

- Unbekannte Ordnung
- braucht gute Hashfunktion

LinkedHashSet - für wenn die Reihenfolge des hinzufügens wichtig ist.

Maps

Maps (od. Dictionaries) speicher key-value Pairs. Auch hier gibt es wieder HashMaps und TreeMaps (analog zu Sets).

Zusammen ergeben List, Set und Map das Collection Framework.

Alle „Ansammlungen" implementieren das Iterator Interface. D.h. sie stellen eine iterator Methode zur Verfügung, die den Iterator selbst liefert.

```
Iterator<T> itr = set.iterator();
```

Solch ein Iterator kann genutzt werden um sicheren Zugriff auf Elemente der Sammlung zu erhalten und mit .remove() das letzte Element sicher zu entfernen.

Klassen & Objekte

Klassen sind „Vorlagen" für Objekte / Programme.
Im Normalfall bieten Klassen einen „Service" an.

Objekte werden durch `Object name = new Object` initialisiert. (Ausnahmen können auch mit Literals initialisiert werden, siehe String).

Reference Semantics and Value Semantics

Reference Semantics bedeutet, dass eine Referenz zu einem Objekt übergeben wird, die den Zugriff auf das Objekt ermöglichen. Arrays!

Value Semantics bedeutet, dass Werte direkt übergeben werden, z.Bsp. Basistypen.

OOP (object-oriented-programming) is a programming paradigm that puts everything into classes. Java is a OOP language, other programming languages like Python or Swift are multiparadigm languages, they dont require OOP, but it can be used.

static provides a method / variable without having a instance of the class.

null is used to let reference variables delete the reference.

Constructors are used to initialize new objects.

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

(name of class)

A class has a default constructor, that init. with null values. One can define as many constructors as needed, but they have to take different parameters. Once a custom constructor is defined, the default constructor is no longer available.

Import von Files

Benötigt java.io.* gibt uns die File Klasse. Ein File Objekt ist aber nur ein Handle, d.h. es kann sein dass das File gar nicht existiert. Deshalb File.exists(). Danach kann ein File mit einem Scanner gehandhabt werden.

Exceptions

Exceptions sind Runtime Errors. Z.Bsp. Division durch 0, File nicht vorhanden etc. Java kann nicht alle Exceptions auffangen, deshalb muss das Programm sie auffangen. Mit throws in der Deklaration einer Methode können wir Exceptions ankünden.

```
public static void name() throws type
```

Der Aufrufer muss die Exception auffangen oder sie weiter ankündigen.

Auffangen geht mit einem try/catch Block.

```
try {
    Code
} catch (exception type  name) {
    Ex. handling
}
```

Das Ankünden einer möglichen Exception geht mit

```
method ... throws  exception type { }
```

oder

```
class  name  throws  exception type { }
```

this

this wird verwendet um auf das Objekt zu verweisen. Damit erlangt man Zugang zu dessen Variablen und Methoden.

this kann auch in einem Konstruktor verwendet werden um einen anderen Konstruktor aufzurufen.

Module

Module sind Teile von Programmen, aber kein vollständiges Programm. Sie besitzen keine main Methode und werden von Klienten verwendet.

Bsp. Math, Arrays, System class.method (parameters)

Access Modifiers

public	zugriff von überall
default	erlaubt nur zugriff von der selben Package
protected	erlaubt nur zugriff von Klasse & Subclasses
private	lässt Zugriff nur von der selben Klasse zu.

Inheritance Ist-ein Beziehung

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

Instanzen der Subclass haben Zugriff auf Attribute und Methoden der Superclass. Eine Subclass kann diese Eigenschaften auch überschreiben `@Override`. Subclasses können überschriebene Methoden der Superclass mit `super.method()` aufrufen.

↑ dies geht nicht bei privaten Attributen

! Wenn die Superclass keinen default-Konstruktor hat, muss die Subclass einen eigenen Konstruktor haben.

Mit `super(parameter)` kann ein Konstruktor der Superclass aufgerufen werden. **Super muss im Konstruktor als erstes stehen!**

Subclasses können Methoden und Variablen überschreiben, welche von der Superclass aufgerufen werden.

↙ werden dann aber nur in diesem Kontext so verwendet!

Wenn eine Superclass als Type verlangt wird, können wir auch alle Subclasses verwenden. **Umgekehrt funktioniert dies nicht!**

Alle Klassen sind Subclasses von der Object Klasse

⚠ „Seitwärts Cast" gibt Runtime Exception ⚠

Interfaces

Inheritance kann nicht alles, z.Bsp. kann nur von einer Klasse geerbt werden.
Daher gibt es eine weitere Art eine „ist ein" Beziehung auszudrücken, ein Interface. Interfaces definieren Methoden die jede Implementierung haben muss, lässt aber keinen gemeinsamen Code zu.

```
public interface name {
    type bar { }
}
```

Alle Implementierungen von diesem Interface müssen bar haben.

Ein interface hat keine Attribute, ausser wenn sie static final sind.

Wenn eine Klasse nicht alle Methoden implementiert gibt es einen Compiler-Error, ausser wir definieren die Klasse als abstract.

Ein Interface kann mehrere Interfaces erweitern.

⚠ Cast zu Interface „geht" immer und gibt ⚠
Runtime Exception

Regeln für guten Code

Damit ein Programm leserlich ist, sollten ein paar Regeln eingehalten werden:

- höchstens 80-100 Zeichen pro Zeile
- eine Anweisung pro Zeile
- wenn ein Ausdruck zu long ist, nach dem Operator trennen & einrücken

```
a + b +
    c + d
```

- Blöcke einrücken (2/3/4 Leerzeichen od. Tab)
- Leerzeichen trennen Keywords

```
for (...)  ✓      myMethod (...)  ✗
while (...)  ✓    x = y  ✓
```

- NewLine zum strukturieren, unterteilt in Gruppen

- {...} immer verwenden, auch wenn nicht nötig
- { auf gleicher Zeile wie if, for, etc.
- Inhalt zwischen Klammern einrücken

Datei Layout:

1. imports
2. main class
3. constants
4. main function
5. additional methods
6. additional classes

Namen:

- _, $ sind reserviert für Spezialfälle
- keine Umlaute

- CamelCase
- Klassen beginnen mit Grossbuchstaben
- Methoden beginnen mit Kleinbuchstaben
  - wenn möglich Verb
- Variabeln beginnen mit Kleinbuchstaben
  - beschreibend
  - kurze Loopcounter (i,j,k)
  - keine Typinformationen
  - Masseinheit im Namen

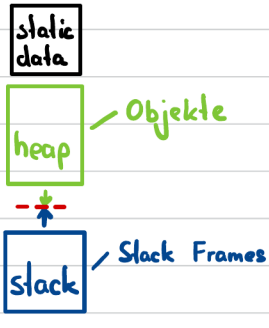
- Konstanten mit Grossbuchstaben

Nachdem erstmaligem schreiben eines Programmes, sollte dieses faktorisiert werden.

Faktorisierung:

- elimination von Redundanz
- mehrfach berechnung vermeiden
- lange Methoden aufteilen
- Methoden standart mässig privat
- Zugriff auf Attribute über getter/setter

Speicher & Adressen



Ein Java Program besteht zur Runtime aus static data, heap und stack.
Methoden haben einen Stack Frame und verwenden dort nur Referenzen zu Objekten.

Adresse

direkte Adresse des Objekts im heap

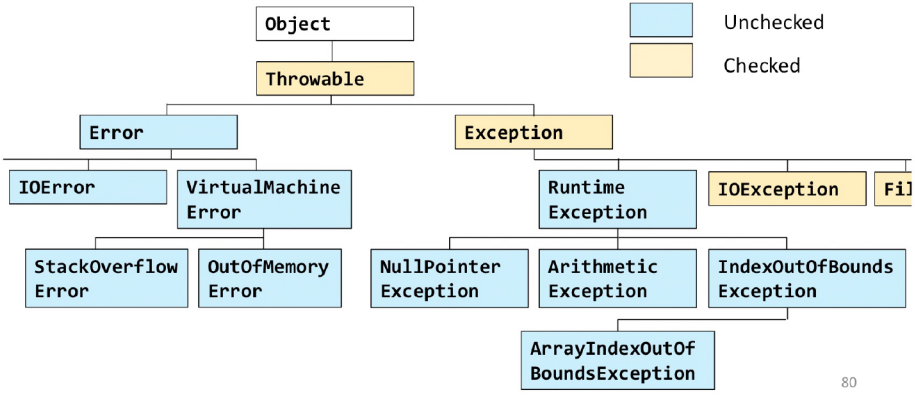
Pointer

Datenstruktur die Adresse des Objekts im heap beinhaltet.

Errors & Exceptions

Error & Exceptions sind Objekte mit der Superclass Throwable. Sie sind dafür da um bei Fehlern während der Ausführung den aktuellen Status festzuhalten und für einen geordneten Shutdown zu sorgen.

Vererbungshierarchie im Einsatz



Dabei gibt es Checked und Unchecked. Unchecked Exceptions müssen nicht angekündigt werden, während Checked Exceptions (z.Bsp. FileNotFoundException) vorher angekündigt werden müssen.

Pre- & Postconditions

Wir können an verschiedenen Stellen in einem Programm Aussagen machen über dessen Zustand. Solch eine Aussage kann immer, manchmal oder nie wahr sein.

Eine Precondition ist eine Aussage, die vor der Ausführung gilt und eine Postcondition gilt danach (unter annahme der Precondition).

Bsp
$\{x < 59\}$
$y = 20$
$\{x + y < 79\}$

Mit solchen Pre- & Postconditions kann eine Hoare Triple gebildet werden.

$\{P\}\ S\ \{Q\}$

S: Programmsegment

Es gibt stärkere und schwächere P/Q, $P1$ ist stärker als $P2$ wenn $P1 \Rightarrow P2$. Oftmals suchen wir die schwächste Precondition $wp(S, Q)$.

Wollen wir solche HoareTriple mit Schlaufen verwenden, so müssen wir auch eine loop-Invariante festlegen.