FOP

Table of Contents

Grundlagen 3

o (FopBot-Kram) 3

o Vorgehensweise 3

o Designhinweise 3

o Java-Theorie 3

o Visualisierungen 6

o Bestandteile von Java Code: 7

o Primitive Datentypen 7

o Arrays 8

o Wrapper-Klassen 8

o Strings 8

o Klassen 8

o Vererbung 10

o Interfaces 11

o Funktionale Interfaces 11

o Enumeration 12

o Record Classes 12

o Control Flow Statements 13

o Operatoren 13

o Statements 14

o Schlüsselwörter 14

o Basisfunktionalitäten 14

o Javadoc 14

· Racket 14

o Theorie funktionales Programmieren 14

o Syntax 15

o Entwurfskonzept 17

· Fehlerbehandlung 18

· GUI 19

o (Applet) 19

o Graphics 19

· Mathe 20

· Random 20

Paused on 6

# Grundlagen

## (FopBot-Kram)

* Raster von unten links, x, y beginnend 0
* new Robot(int x, int y, Direction d, int NumberOfCoins)
* move()
* turnLeft()
* putCoin()
* hasAnyCoins()

## Vorgehensweise

* Aufgabe formulieren
* Grundshema, Teilprobleme formulieren, Gemeinsamkeiten finden, Lösungen mehrfach nutzen
* Schleifen desginen:
  + Invariante: In jedem Schleifendurchlauf gleich
  + Variante: In jedem Schleifendurchlauf anders
  + Konsequenz: Daraus resultierendes Ergebnis
* Refactoring

## Designhinweise

* Werten immer Namen geben (Variablen/Konstanten statt konkrete Zahlenwerte im Code) (Änderbarkeit und Lesbarkeit)
* Konstanten wo möglich (Schutz vor versehentlichen Änderungen)
* Sonderfälle beachten
* Konvention: get/set Attribut großgeschrieben (nicht zwangsläufig interner Name oder wirklich ein Attribut) (für Abkapselung und leichtere interne Änderungen)
* Randfälle testen

## Java-Theorie

* Objekt im Datenspeicher header mit offSet der Attribute
* Quelltext, (ausführbarer compilierter) Bytecode
* Compilerfehler (Syntax), Laufzeitfehler (Teilen durch 0 (außer double), null pointer, index out of bounce, Speicher voll, Call-Stack voll))
* Model Programm Counter: Adresse der nächsten auszuführenden Anweisung, diese wird in jedem Taktzyklus in die CPU geladen
* Deklarieren, Initialisieren
* Referenzvariablen (von Referenztypen) <-> primitive Datentypen   
  (primitive Datentypen werden z.B. auch bei Methodenaufruf kopiert)
  + Referenztypen: Klassen, Interfaces, Arraytypen, Enum-Typen, Record Classes (ungefär was nicht in Java eingebaut)
  + Unterschied zwischen Referenz und Objekt
  + wichtige Grundlage für Objektorientierung
  + Überschreiben der Referenz auch „verbiegen“ genannt
  + Obejktidentität und Wertgleichheit
    - Wertgleichheit erfordert Wertgleichheit aller Attribute (rekursiv))  
      (Achtung z.B. String) (Objektidentität auch Wertgleichheit?)
  + Deep <-> Shallow Copy (meist deep copy gewünscht)
* Nur Speicherplatz ist Objekt
* Konstanten: Attribut mit Keyword final unveränderbar (muss initializiert werden)
* Überlauf ohne Fehlermeldung
* Bei gebrochenen Zahlen Genauigkeitsprobleme (statt Test auf Gleicheit, Test auf ausreichend nahe beieinander)
* nur eine einzige Definition in einer Quelldatei darf public sein und muss der Name der Datei sein
* Zugriffsrechte:
  + Definitionen public oder gar nichts
  + nur eine public pro Datei (muss nicht an erster Stelle sein) (z.B. kein Modifier ist zusätzlich erlaubt)
  + private: nur in der Klasse selbst (auch andere Objekte dieser Klasse)
  + implizit (nichts): zusätzlich zu private auch im Package
  + protected: zusätzlich in allen abgeleiteten Klassen
  + public: überall wo Package importiert
* Ausführung von Methoden (FOP 01e):
  + Speicherbereich (Frame) wir für Daten der Methode reserviert für
  + Referenz zu dem Objekt mit dem es aufgerufen wurde (in jedem Frame bestimmte Adresse dafür)
  + Rücksprungadresse
  + jede Information hat dann einen gewissen offset, der dann auf den Stackpointer aufaddiert wird
  + nach Beendigung wird der Frame wieder von dem Call-Stack geholt
  + return Wert wird in unbennate Stelle (typischerweise bestimmtes Register) geschrieben und kann von da ausgelesen werden
* Stacktpointer zeigt auf Element im Call-Stack (Stack der Frames der Methoden)
* main richtet sich selber ein, nach ablegen von main Prozess beendet
* Methoden werden falls nicht anders automatisch mit eigenem Objekt aufgerufen
* Unterschied Attribut, Parameter, Variable
  + Scope (Ort der Gültigkeit der Identifier (für Methoden, Typen, Variablen, …)
  + Scope ist statisch, daher eindeutig von Quelltext
  + Typpen und Methoden: Siehe auch zugriffsrechte (auch über Einführung des Identifiers)
  + Parameter: ganzer Methodenrumpf
  + Lokale Variable: ab Einführung bis Ende des Bereichs des innersten umfassenden Klammerpaars (Ausnahme for-Schleife inkludiert ()), aber lesend erst nach erstem schreibendem Zugriff, da Anfangs zufällig Initialisierung (nur lokale Variable)  
    (Ausnahme try-catch, …)
  + Parameter und lokale Variablen werden vor Attributen angesprochen, Zugriff auf Attribute müssen bei Namensdopplung mit Klassen/Objektangabe spezifiziert werden)
* Bei Namenskonflikt, Parameter vor Attribut (siehe Konstruktor)
* Packages
  + Zusammenfassung mehrerer Definitionen
  + package name; //Gibt Zugehörigkeit an (muss erste Zeile sein, nur ein Packages, dieses wird auch importiert)
  + import (immer am Anfang der Datei)
    - import static (Klasse kan ohne voranschreiben der Klasse verwendte werden, kann auch für ein spezifisches Klassenattribut genutzt werden)
    - .\* alles aus diesem Package (aber keine Subpackages)
  + Package Unterstruktur mit . getrennt
  + Typ kann auch ohne import durch voll qualifizierten Namen angesprochen werden?
  + Voll qualifizierte Name einer Klasse mit Package.Klasse
  + Java.lang wird automatisch importiert
  + Für Struktur, Auffindbarkeit, protected, …
  + Namenskonflikte kontrollierbar
    - Keine zwei Typen (Definitionen) mit gleichem Namen in einem Package
    - Bei unterschiedlichen Namen, beide mit Packagenamen vorweg ansprechen, sonst Compilerfehler
  + Weltweit einheitliche Namenskonvention für Packages, da zwei gleichnamige nicht importierbar
    - Firmen und Institutionen (umgedrehte Domain-Name, Underscore statt unzulässige Zeichen)
  + Module seit Java 9 Zusammenfassung von Packages zur besseren Steuerung, nicht in FOP behandelt
    - Modul java.base automatisch importiert (Hausübung 10 importiert java.desktop)
* Java-Standardbibliothek (Teil jeder Java-Distribution)
* formale Parameter (Definition von Typname und Identifier)  
  aktuale Parameter (Wert des Parameters in einem Aufruf (muss impliziert in formallen konvertierbar sein))
* Garbage Collector
  + Teil des Laufzeitsystems (Aufruf nach gewissen Regeln)
  + Konfiguration zur Optimierung möglich
  + Gibt Speicherplätze frei, die vom Programm nicht mehr erreichbar sind
* public static void main(String[] args){} Einstieg ins Programm args odt Kommandozeilenparameter genannt, Angabe bei Aufruf über Kommandozeile Trennung mit Whitespaces
* (offizielle) Begriffsbildung Oracle Docs
* Ausdrücke
  + Rechtsausdrücke (rvalues): haben Typ und Wert (kann rechte Seite einer Initialisierung, Zuweisung, aktualer Parameter, Rückgabewert oder arrayindex sein (array[rvalue], muss impliziert in int konvertierbar sein), sowie rekursives Zusammensetzen z.B. (n+2)<5
  + Linksausdrücke (lavalues): Verweise auf Speicherstellen (linke Seite einer Initialisierung, Zuweisung, Arraykomponenente, Attribut) (diese hat auch Typ und Wert?)
* Ausdrücke haben neben Typ und Wert optionale Seiteneffekte
  + z.B. ++n Seiteneffekt inkrementiert n, Wert von n danach  
    n++ Seiteneffekt inkrementiert n, Wert von n davor
  + Zuweisungen haben auch Wert (z.B. a = b += c)
  + (void Methode ohne Seiteneffekt hat keinen Effekt) (Rückgabe quasi „Haupteffekt“)

## Visualisierungen

* Flußdiagramm als visuelle Darstellung von Control Flow Statements
* Prozessmodell (durchnummerierte Schritte, mit Spüngen)
* (Tabelle der Attributwerte der einzelnen Durchläufe einer Schleife)
* UML-Klassendiagramm
  + Klassen als Kasten mit drei Teilen: Name, Attribute, Methoden (incl. Konstruktor)
  + - private, # protected, + public
  + name(name: datentyp, ...): Rückgabetyp
  + Vererbung durch Pfeil mit Dreieck zur Basisklasse (auch bei Vererbung zwischen Interfaces)
  + Strich Assoziation (kann Annotiert werden (auf Seite zu der ein Pfeil zeigen würde)(egal: 0..\*, min 1: 1..\*, 1, ...) (außerdem wird das Attribut der Assoziation mit zugriffsrecht drangeschrieben)
  + <<interface>> über Name für Interface
  + implementiert durch gestrichelten Pfeil mit Dreieck zum Interface

## Bestandteile von Java Code:

* Keywords (nur an bestimmten Stellen, hat bestimmte Funktion)
* Identifier (Bezeichner von Entitäte, (teilweise) selbst gewählte Namen (oder Bibliothecken))
  + Bennenungsregeln
  + ‚a‘-‘z’, ‘A’-‘Z’, ‘0‘-‘9‘, ‘\_‘, ‘$‘, weitere ...
  + Erstes Zeichen keine Ziffer!
  + Keine Keywords!
  + Konventionen: Klassennamen groß, Attributnamen klein, CamelCase, Konstanten komplett Großgeschrieben
* Klammerpaare: (), {}, [], <>
* Kommentare: //Rest der Zeile, /\*Kommentar\*/, /\*\*Java Doc \*nächste Zeile [\*@tag](mailto:*@Tag) ... \*/
* Whitespaces: blank, newline, tab
  + nicht innerhalb eines lexikalischen Elments
  + min. Eins zwischen zwei schlüsselwörtern, Identifiern
  + ansonsten ignoriert
* Literale (wörtlich geschriebene Werte eines Datentyps)   
  z.B. „abc“ (String), null (symbolischer Wert)

## Primitive Datentypen

* GanzeZahlen: byte (8 Bits), short (16), int (32), long (64) (twocomplements)  
  123, +123, -123 (auch für kleinere genutzt), 123l, 123L (für long)  
  Nullwert 0
* Gebrochene Zahlen: float, double  
  12.34, .123, 1.2E-34 (gebrochenzahlige Literale (double)), .12f, .12F (float)
* Logik: boolean   
  ( true, false)  
  Nullwert: false
* Zeichen: char (Unicode) (intern auch Ganzzahl)  
  ‚a‘, Unicode mit Nummerf in Hexadezimal ‚\u03A9‘, ‚\\‘ für \, ‚\‘‘ für ‚ ‚\t‘ tabulator  
  Nullwert ‚\u0000‘ (steht für kein Zeichen)
* Ergebnistyp:
  + bei zwei ganzzahligen (außer long) immer int (außer in für den Compiler einfachen Außnahmen)
  + sonst immer das gößere der beiden Operanden: int -> long -> float -> double
* Konversionen
  + implizit nach oben
  + explizit (z.B. int .123, aber wenn nicht darstellbar Murkswert))

## Arrays

* Arrays are Objects: Classenname[], new Classenname[length] / Classenname {Elemente, ...}
* Index ansprechen: Classenname[i]
* Hat Konstante length (fixed length)
* auch von primitiven Datentypen
* Arrays von Arrays
* zu jedem Typen gibt es einen Arraytypen
* Komponententyp können primitive Datentypen sein  
  oder die Komponenten sind Referenzen auf Objekte des Komponenttyps
* dynamische Typ einer Variable eines Arrays kann auch ein Array von Subtypen sein  
  Subtypen eines Arrays mit Komponenttyp T sind alle Arrays mit Komponententypen, die Subtypen von T sind  
  (Subtyp Relation überträgt sich vom Komponententyp)
* Array von Referenztypen automatisch mit null initiallisiert

## Wrapper-Klassen

* Zu jedem primitiven Datentyp eine groß geschriebene Wrapper-Klasse (außer Character for char, Integer for int) (in java.lang)
* Haben Konstruktor mit entsprechendem Datentyp als Parameter, sowie entsprechende .primDatentypValue (z.B. bo.booleanValue())
* Außerdem haben sie z.B. statische Attribute (z.B. Double.MAX\_VALUE)
* Implizites Boxing und Unboxing

## Generics

* Klassen können generisch, durch Typparameter parametrisiert sein
  + Public class name<Typparameter1, …>{/\*…\*/}
  + Um instanziierten Typ zu erhalten: z.B. List<Inetegr> (gesprochen List ist mit Integer instanziiert, Liste von Integer) Bei Konstruktor, wenn erschließbar sind die Typparamter weglasbar, nicht aber <> (oft Diamond-Operator genannt, aber kein Operator)
  + Klassenmethoden sind nicht von Typparamtetern der Klasse betroffen
* Auch interfaces und record-Klassen können generisch sein
* Generische Methoden
  + Vor Rückgabewert (mit Whitespace) Typparameter in spitzen Klammern (damit Rückgabewert schon parametrisiert sein kann)
  + Bei Ausführung wird Typparameter aus Kontext erkannt
* Typparameter können als statischer Typ genutzt werden (z.B. T1 attribut oder als Rückgabewert)
* Auch Arrays als Typparameter erlaubt (z.B. int[], String[])
* Nichtübertragbarkeit von Vererbung auf Typparamter für List<X> kann kein List<Y> verwendet werden (Y extends X)
* Einschränkung von Typparametern
  + <T extends X> (T ist durch X beschränkt, X kann auch Interface sein)  
    mehrere: <T extends A & B> (ist eine Klasse dabei, muss sie als ersets kommen)
* Wildcards
  + Einschränkung von Instanziierungen von Typparametern (als Parameter, ist nur der einzelne Parameter generisch, nicht die ganze Methode)
  + X<?> können alle Instanziierungen von X sein (wie X<? extends Object> (Upper Bounded Wildcards)  
    Empfohlen für In/Out-Parameter und Rückgabewert
  + In X<? extends Number> kann auch ein X<Integer> gespeichert werden  
    Empfohlen als In-Parameter (z.B. List<? extends Number> list list.get().doubleValue())
  + In X<? super Double> (Lower Bounded Wildcards)  
    Empfohlen als Out-Parameter (z.B. (List<? super Number> list){list.add(1);}
* Einschränkungen, da Java Bytecode keine Typinformationen enthält und Generics erst später ginzugefügt wurden (Oracle-Documentation Artikel „Restrictions on Generics“)
  + Keine primitiven Datentypen
  + Keine Erzeugung von Objekten oder Arrays
  + Keine Klassenattribute von Typparametern
  + Kein Downcast oder instanceof mit Typparametern
  + Kein throw-catch mit Typparametern (instanziierte Exceptions können nicht geworfen werden)
  + Keine Methodenüberladung mit Typparametern

## Strings

* java.lang.String
* Sequenz von chars
* anders als Array nicht fest eingebaut und kein primitiver Datentyp, aber:
  + Literale (Menge von char Literalen mit „“ statt ‚‘ bei chars)
    - in neueren Jaav-Versionen „““white Spaces werden Übernommen„““
  + + Operator funktioniert wie str1.concat(str2) (erstellt neues Objekt) (+ mit primitiven Datentypen ergibt String)
* Methoden: new String(„“), str.length(), str.charAt(i), str.indexOf(char c), str.matches(„regulärer Ausdruck (. beliebiges zeichen, + Zeichen davor ein oder mehrmal“)
* nützliches: System.getProperty(„line.seperator“)

## Klassen

* Kopf der Klasse: public [static] [final, sealed, non-sealed] class Name [extends ...] [implements ...] [permits ...] { Rumpf }
* Konstruktor: zugriffsrecht Klassenname(Parameter) [throws ..., ...]{Methodenrumpf (Sequenz von Anweisungen)}  
  Suche Speicherplatz, reserviere diesen, Konstruktor aufrufen, Adresse zurückliefern  
  Objektkonstanten auch in Konstruktor initialisierbar  
  muss super(...) aufrufen, es sei denn unmittelbare Basisklasse hat Konstruktor ohen Parameter  
  wenn nicht definiert Default Constructor (Teilweise auch eigener Konstruktor mit leerer Paramteterliste)
* Attribute: Klasse/Datentyp name; (Können auch bei Deklaration initialisiert werden, sonst Nullwert (0/‘\u0000‘/null))  
  (auch mehrere Definitionen auf einmal möglich z.B. int m=1, n, k=2;)
* Methoden:
  + Header: ungeordnete Modifier Rückgabetyp Identifier Parameterliste ungeordnete Exceptions
  + Rumpf: {/\*Anweisungen\*/}
  + Zugriffsrecht [static] [final] Rückgabetyp name(Parameter){Methodenrumpf} (return beendet Methode, geht auch bei void)   
    (Überladen Methoden, die sich nur durch Parameter unterscheiden (evtl. Zusätzlich im Rückgabewert) (oft für Standardwerte)(Signatur muss sich unterscheiden)  
    Problem bei (int i, double d), sowie (double d, int i) bei Verwendung mit uneindeutigen Literalen Compilerfehler)  
    (Alles vor Rückgabetyp sind ungeordnete Modifier)  
    Parameterliste: (geordnete Sequenz) nach außen nur Typen und ihre Reihenfolge wichtig  
    Modifier final: Methode kann nicht überschrieben werden (empfohlen für Methoden, die der Konstruktor aufruft)
  + Signatur einer Methode: Name, Parameterliste (geordnete Sequenz der Typen)
  + variable Parameterzahl (nur am Ende der Parameterliste): (..., Typ... identifier) aus Typ... wird dann ein Array vom Typ mit beliebiger Länge (auch 0) (aber niemlas null)  
    Aufruf mit belibig vielen Attributen des Typs oder mit array<
* interne Umsetzung:
  + annonymes Objekt der Klasse auf das ein anonymes Attribut im Header der Objekte von ihr verweisen
    - mit Informationen: zur Klasse selbst, den Attributen, den Methoden,   
      (jede Methode (int Methodentabelle) mit einem bestimmten Offset, der sich bei vererbten Methoden nicht ändert (wesentlicher Grund gegen Mehrfachvererbung)) (auch Attribute haben festen Speicher offset in Objekten)
    - sowie Methodentabelle (mit Referenz auf Implementation der Methode (diese kann bei Verterbung überschrieben werden)
  + Methodenaufruf: Referenz auf Objekt, auf annonymes Objekt der Klasse auf Methodentabelle auf Methode (da offset identisch, kann er beim compilieren als Konstante eingesetzt werden)
* static:
  + static Initializer: static{/\*Anweisungen\*/} (wird zur Begin des JavaProgramms ausgeführt)
  + Klassenattribute: statische Attribute (ansprechen über Klasse oder Objekte der Klasse)
  + Klassenmethoden: statische Methoden (dürfen nicht auf Objektattribute oder) Objektmethoden zugreifen
* abstract class
  + kein Objekt mit new erzeugbar
  + kann abstract Methoden ohne Methodenrumpf (auch ohne {}) haben
* Keywords: this (eigenes Objekt), super (unmittelbare Basisklasse)

## Vererbung

* extends in Klassenkopf
* Alle Klassen (die nicht von anderer abgeleitet) impliziet von java.lang.Object abgeleitet (nicht bei Interfaces)  
  hat: obj.eauals(Object other), toString() (tun nicht unbedingt wofür sie gedacht sind)  
  (nicht in FOP: professionell ist auch die Funktion hashCode relevant)
* Basisklasse vererbt alle Attribute und Methoden (auch Klassenmethoden)
* Konstruktoren werden nicht vererbt, erster Konstruktor Aufruf muss super(...) sein (wenn keine Parameter, automatisch von Compiler ergänzt) (nicht vererbt, da der Konstruktor nur ein Objekt der Basisklasse ienrichten würde und den Subtyp vermutlich nicht richtig initialisiert)
* zusätzliche definierbar
* Attribute des gleichen Namen (dank super) möglich, aber nicht empfohlen
* Methoden überschreibar selber Kopf außer Sichtbarkeit kann höher werden)  
  auch Methoden der Basisklasse rufen Methoden der Subtypen auf  
  Signautr der Methoden muss gleich sein und die anderen Bestandteile haben Grenzen (Sichtbarkeit sichtbarer machen, Subtyp des Rückgabewert, Subtypen der Exceptions)  
  Begriffe: überschreibende Methode überschreibt überschriebene Methode
* super keyword for Basisklasse (only next highest)
* überall wo ein Referenztyp erwartet wird, kann auch ein Objekt eines Subtyps verwendet werden
* Begrifflichkeiten: Supertyp und Subtyp (in FOP: Referenztyp ist nicht sein eigener Super/Subtyp)
* statischer Typ (Typ einer Referenz) bestimmt welche Methoden aufgerufen werden können  
  dynamischer Typ (Typ des Objekts (der Referenz)) bestimmt welche Version der Methode aufgerufen wird  
  formal gibt es auch den dynamischen Typ Nulltyp  
  Achtung bei Klassenmethoden bestimmt der statische Typ über Methodenimplementation (da es bei Aufruf mit Klasse ja keinen dynamischen Typ gibt)
* Downcast: „gefährliche“ Konversionen müssen explizit erfolgen (Typ) objekt (man sollte davor Operator instanceof nutzen)
  + Kurzform: (a instanceof X b) ... b.m1() (genannt „patern matching“)
* final class/interface: keine Klassen ableitbar
* sealed class/interface [extends ... implements ...] permits Subtyp1, ...{} erlaubte Klassen müssen im gleichen Package sein  
  Subtypen müssen auch final, sealed oder non-sealed (letzeres bricht diese Pflicht explizit auf) sein

## Interfaces

* Public interface Name [extends otherInferace1, otherInterface2, ...]{}
* Keine Attribute
* deklarierte Objektmethoden (immer public (üblich public wegzulassen, weil sowieso)) (ohne Methodenrumpf, sondern mit Semikolon)
* Implementierte Objektmethoden in Form von Default-Methoden (implizit public)  
  default rückgabetyp name(param){/\*Anweisungen\*/}  
  werden zwei default methode gleicher Signatur vo Interfaces implementiert muss die methode überschrieben werden, sonst Fehlermeldung
* Implementierte Klassenmethoden (implizit public)
* Klassenkonstanten (implizit public und final)
* kann statischer Typ sein
* Eine Klasse kann beliebig viele Interfaces (mit implements InterfaceName1, ...) implementieren (muss dann Methoden implementieren)  
  erfordern zwei Interfaces die gleiche Methode mit anderem Rückgabetyp, können nicht beide importiert werden
* final und selaed wie bei Klassen
* (in Package java.util.function mehrer mathematische Interfaces mit apply)

## Funktionale Interfaces

* Konzept der funktionalen Programmierung Funktionen als Daten
* Interface mit genau einer Methode, die nicht default oder static ist (funktionale Methode)
* Literale (Lambda-Ausdrücke) (Compiler richtet namenlose nicht sichtbare Klasse ein) (eigentlich kein eigenes Sprachkonstrukt, sondern Abkürzung)
  + Fachbegriff Closure: Informationen aus dem Entstehungskontext des Lambda-Ausdrucks werden mitgespeichert, aktuale Wert wird nicht unbedingt kopiert (sondern eventuell referenziert), daher nur konstante oder effektiv konstante Werte, sonst Fehler (effektiv konstante Werte (effectively final), sind Werte nicht mehr geändert werden)  
    Bsp. Wert von ... Wird durch closure gespeichert
  + Kurzform: (parametername1, pn2, ...)-> Wert //Typ wird vom Compiler aus dem statischen Typ erschlossen (Bei nicht genau einem Parameter Klammern erforderlich)  
    z.B. IntToDoubleFunction fct = x –> x\*10; double z = fct.applyAsDouble(11);
  + Methodenreferenz: Object::method (z.B. System.out::print Compiler kann erschließen, welche print methode gemeint is, Math::maxt)  
    auch Konstruktor: z.B. String::new (z.B. String[]::new Konstruktor bekommt die Länge))
  + Standardform: (Typ Parameter, ...) -> {/\*Methodenrumpf\*/}
  + Beispiel IntPredicate (java.util.function) at die default Methoden negate, and, or  
    .apply, DoubleToDoubleFunction.applyAsDouble (Rückgabetyp am Ende), IntConsumer.accept, StringSupplier.get()

## Enumeration

* public enum Name { KONSTANTE,... }
* Enums are Classes (Zusammenfassung mehrere Konstanten zu einer Einheit)
* Die Konstanten sind Referenzen zu einem Objekt
* Verwendung: Name variable = Name.KONSTANTE, enum1 == enum2
* Alle abgeleitet von java.lang.Enum (aber Klasse icht auf normale Weise von Enum ableitbar)
* Zu Beginn des Programms werden alle Konstanten als Objekt erzeugt, auf die die Klasse verweist
* keine Vererbung
* Funktionen: Enum.values() (returned Array der Konstanten), konst.name() (String wie in Definition)

## Record Classes

* Vereinfachte Form von Klassen (Sammlung privater Objektattribute mit standardfunktionalität)
* record Name(Typ id, ...){/\*eingeschränkter Klassenrumpf\*/} (definierte Attribute private and final)
* geschenkt:
  + Konstruktor (canonical constructor) mit gleichen Parametern, der die Attribute setzt
  + rec.attribute(); (wie getter)
  + rec.equals(Object obj) (true bei selbem dynamischen Typ, sowie Wertgleichheit und gleicher (dynamischer) Typ aller Attribute)

## Control Flow Statements

* for(initial Anweisung;Fortzetungsbedinung;Fotzsetzungs-Anweisung){Anweisung /\*Schleifenrumpf\*/}  
  Kurzform üfr Arraydurchlauf: for(Typ identifier : Array){/\*Schleifenrumpf\*/}
* while(Fortsetzungsbedingumg){Anweisungen}
* do{Anweisungen}while(Fortzetungsbedingung)
* (anderes Konstrukt) if(Bedinung){Anweisung}else{Anweisung}
* {} kann bei einzelner Anweisung weggelassen werden
* switch(Ausdruck von byte, char, short, int, Enum oder String){case (Wert vom Typ des Ausdrucks): /\*Anweisungen\*/ [break;] [default: ]} (Aber Ausdruck muss zur Kompilierzeit fest stehen, daher nur aus Literalen und Konstanten und arithmetischen Ausdrücken)  
  Zusammenlegung zweier cases: case a: case b: Anweisungen break;
  + ab Java 12 vereinfachte switch Anweisung:  
    switch(Ausdruck){case Wert, Wert, … -> Anweisung; default -> Anweisung;}
  + weiter Vereinfachung (switch Ausdruck)  
    a = switch(Asudruck){case Wert, … -> Wert; …};
  + wird höchstwahrscheinlich Java-Standard  
    a = switch(Objekt){[case null -> returnWert;] case Subtyp identifier -> returnWert; … [default -> returnWert (nicht benötigt, wenn statischer Typ sealed)] }
  + … (Suchwort „pattern matching“)

## Operatoren

Binär (zwei Operanden)

Infix (steht zwische Operanden)

* ++, -- (unär, Präfix und Postfix)
* +, -, !, ~ (unär)
* \*, /, %
* +, - (binär)
* (<<, >>, >>> (Bitverschiebung) (~, ^, |, &, <<, >>, >>>))
* =, +=. -=, \*=, /= (zuweisungsbasierte Operatoren)
* <=, >=, <, > (Vergleichsoperatoren)
* ==, !=
* Bitweises & -> ^ -> |
* logisches && -> ||
* (Bedingung)? (Wert, wenn true): (Wert, wenn false) (tenärer Bedinungsoperator)
* new, instanceof (Bindungsstärke?)

nach Bindungsstärke aufsteigend (transitiv):

## Statements

* break (beendet (tiefste) Schleife), continue (beendet aktuellen Schleifendurchlauf)

## Schlüsselwörter

* final (bei Deklaration, macht Konstante)

## Basisfunktionalitäten

* Integer.MAX\_VALUE, Double.POSITIVE\_INFINITY, Double.NaN
* Thread.sleep(delay (millisekunden))
* java.lang.System
* java.math.BigDecimal beliebig genaue Zahl
* Calendar.getInstance().get(Calender.HOUR\_OF\_DAY)
* Java.lang.Number (Superklasse der Wrapper-Klassen außer char und boolean)
* Java.util.Comparator<T> Interface mit int compare(T t1, T t2) (0 wenn gleich, negativ, wenn erstes kleiner, positiv, wenn erstes größer)
* Comparable Interface int compareTo

## Javadoc

/\*\*

\* Beschreibung

\*

[\*@Tag1](mailto:*@Tag1) ...

[\*@Tag2](mailto:*@Tag2) ...

\*/

Tags:

* Typischerweise ganze Quelldatei:
  + @author
  + @version
* Methoden:
  + @throws class Klassenname Bedingung, wann sie geworfen wird
  + @param name Beschreibung, [Vorbedinungen]
  + @return Beschreibung

# Racket

## Theorie funktionales Programmieren

* zwei grundsätzliche Modelle für Abstraktion (Paradigmen)
  + Java vorrangig durch objektorientierte Paradigmen geprägt
    - imperativer Programmierstil (momentaner Zustad und zeitliche Abläufe)
    - Objekte und Klasse zu denen alle Subroutinen gehören
      * Design: Zerlegung in Klassen und Interfaces, Konzepte = Klassen und Interfaces, Ablauf = Interaktion von Objekten, Ererbung erweitert, verfeinert oder variiert
    - Statische Typisierung (Prüfung durch Compiler) (mehr Fehlersicherheit, bessere Laufzeit)
  + HtDP-TL als Dialekt von Racket als Dialekt von Scheme
    - deklarativer Programmierstil (kein zeitlicher Ablauf) (nur „Formel“ des Ergebnisses)
    - Funktionen zentrale Bausteine (primäre Entitäten) (f: D1 x ... x Dn -> R n-Inputs (Definitionsbereich Domain) ein Output Wertebereich (Range))
    - Design: Zerlegung in Funktionen, die sich gegenseitig aufrufen, Variationen durch Funktionen als Parameter
    - gleiche Parameter -> gleiches Ergebnis (keine Seiteneffekte) (Fachbegriff: referentielle Transparenz)
    - idealisiertes Objektmodell
      * nur Konstanten, immer kopieren des Wertes (Laufzeitsystem kann zur Optimierung abweichen) (keine Objektidentität)
    - Dynamische Typisierung (Typ wird zur Laufzeit geprüft) (kürzerer Quelltext angeblich schnellere Entwicklung)  
      (Es gibt auch welche mit statischer Typisierung, trotzdem muss der Typ nicht hingeschrieben werden, sondern wird erschlossen) (Fachbegriff Typinferenz für wenn die Typen nicht zusammen passen)

## Syntax

* Ergebnis wird automatisch in Ausgabefenster geschrieben
* keine Typparameter (Typfehler Fehlermeldung bei Laufzeit)
* Klammern genau um jede syntaktische nicht atomare Einheit (keine Bindungsstärke)
* Präfixnotation bei Operatoren
* Kommentare Konvention:
  + Konvention ganzzeilige Kommentare mit ;;
  + Über Methode „Vertrag“ der Methode
  + ;; Type: number number -> number (z.B. auch (list of ANY), (number -> number) for a fct) or ;; Type: X->X für beliebigen Typen, der dann auch das Ergebnis ist (großes X problemlos nutzbar, da nach Konvention alle Identifier klein geschrieben)  
    [;; Precondition: the secound parameter must not equal zero]  
    ;;  
    ;; Returns: the sum oft he two parameters  
      
    (define (add x y) (+ x y))
* Identifier regeln:
  + Verboten: ( ) [ ]{ } „ , ´ ‚ ; # | \ Whitespaces
  + Nicht nur Ziffern
  + Rest erlaubt (schon belegte werden überschrieben)
  + Konvention: keine Großbuchstaben, Bindestriche zwischen Wörtern
* Zahlen
  + Ganzzahlig, rational (z.B. Literal: 3/5), nichtexakt (z.B. (sqrt 2), komplexe Zahlen (z.B. 1+1i)
  + Für Programmierer möglichst einheitlich
  + Nicht exakt darstellbare Zahlen werden mit #i davor ausgegeben
* String Literale mit „“ (in Racket eingebaut (nicht wie in Java Klasse))
* Symbole: ‚Identifier (steht für nichts außer sich selbst)
  + Mögliche Operationen (symbol? wert), (symbol=? symbol1 symbol2)
* Gibt auch Arrays
* Listen
  + Erstellung: (list elemente) (ist eine Funktion), empty (vordefinierte Konstante)
  + (cons element liste) (Ergänzung vorne), (first name), (rest name), (empty? Name)  
    (Außerdem (second name), ..., (tenth name)
  + Homogen, wenn alle Elemente vom selben Typ, sonst heterogen
* Structs
  + (define-struct name (feld1 feld2 ...)) (Wir nennen die Felder auch Attribute)
  + (make-name wert-fuer-feld1 ...) (alle müssen abgegeben werden)
  + (structname-attributename struct) z.B. (student-last-name student1)
  + (structname? wert) ;Test ob vom Typ
* Funktionen als Werte
  + z.B. (define add +) (add 2 3), oder z.B. auch in structs
  + Funktionen höherer Ordnung sind Funktionen mit Funktionen als Parameter
  + Lambda-Ausdrücke: (lambda (parameter ...) wert)
* #<void> Literal vergleichbar zu null in Java
* (Schlüsselwort Parameter1 Parameter2 …)
  + (define (identifier Parameter) Ergebniswer) Aufruf: (identifier Parameter)
  + (define konstantenname wert)
  + ;Rest der Zeile Kommentar
  + (local (;Definitionen) ;Ausdruck) ;Wert des Ausdrucks ist Wert von local
  + (begin ausdruck ausdruck …) letzter Wert ist der Wert von begin
  + (set! Konstantenname wer) überschreibt Konstante mit Wert (keine Rückgabe)
* Arithmetische Operationen
  + +, -, \*, / (nicht unbedingt binär z.B. (- 1 2 3) entspricht (- 1 (+ 2 3)), (+ 1) ist 1, (/ 2) ist 1/2)
  + Eingebaute Funktionen: modulo, sqrt, floor, ceiling (aufrundne), gcd (größter gemeinsamer Teiler)…
  + Konstanten: pi, e
* Boolesche Operationen
  + Literale: #t, #f
  + Operatoren: and, or, not
  + Operatoren mit booleschem Ergebnis:
    - =, <, <= z.B. (< 1 2 3) ist (and (< 1 2) (< 2 3))
    - number?, real?, rational?, integer?, natural?, symbol?, string? (nur unär)   
      (z.B. (integer? (+ 1/2 1/2)) ist #t, (natural? 0) ist #t
    - (eq? Test auf Objektidentität, aber da freie Systementscheidung nicht empfohlen)
* Control Flow
  + (if boolscherWert ergebnis-wenn-wahr ergebnis-wenn-falsch)
  + (cond [boolean wert] [boolean wert] [else wert]) (falls kein else und kein Fall zutrifft Fehler)
  + (error message)
* Laufzeitchecks:
  + (check-expect wert wert)
  + (check-within wert min max)
  + (check-error (berechnung) fehler) z.B. „/: division by zero“
* Konzept Rekursion (Selbstaufruf einer Methode)
  + Abbruchsbedinung (sonst Endlosschleife)
  + Bsp. Mathe Fibonacci, Pascalsche Dreieck
  + Grundlegendes Konzept, da Schleifen funktionalem Konzept widersprechen
* Konzept Akkumulator (akkumuliert)
  + Bsp. list-of-sums with the local function list-of-sums-accu that also gets a accu as parameter (aufaddieren von Listen Elementen)
  + Akkumulator Speicher für Zwischenergebnisse?
* Standardfunktionen
  + filter ;; Type: (X -> boolean) (list of X) -> (list of X)  
    Entfernt alle Elemente, die nicht dem Prädikat entsprechen
  + map ;; Type: (X -> Y) (list of X) -> (list of Y)  
    (heißt auch Projektion, X wird auf Y projiziert))
  + foldl (von links nach rechts) und foldr ;; Type: (X Y -> Y) Y (list of X) -> Y  
    Mit Initialwert komulieren aller Listenelemente  
    foldl: linksassoziativ, foldr: rechtsassoziativ

## Entwurfskonzept

* Top-down schrittweise Verfeinerung, in Funktionen aufteilen (an Funktionen delegieren), die man dann später implementiert (on the fly structs designen) (wieder lieber Konstanten als Werte direkt reinschreiben)

# Fehlerbehandlung

* Laufzeitfehler (nicht von Compiler, Programmabruch, Ausgabe Call-Stack) (z.B. für durch Sprache ausgeschlossene Laufzeitfehler teilen durhc 0 bei double durch \*\_INFINITY und NaN)
* Exception Klassen sind alle Subtypen von java.lang.Exception (Konstruktor mit String message and getter) (für debugging exc.printStackTrace();) (häufig wird eigene abgeleitet)
* Mechanik Exception
  + throw objektVonThrowable; (muss in Methodenkopf durch throws Klasse1, … (throws-Klausel) gekennzeichnet werden, beim Überschrieben von Subtypen, kann es spezieller werden)  
    Methodenausführung wird beendet
  + try{/\*Anweisungen try-Block\*/}  
    catch(ExcKlasse identifier [ | Klasse2 id2]){/\*Anweisungen catch-Block\*/}  
    catch(…){…}…  
    [finally{/\*Anweisungen\*/}]  
    wird eine Exception geworfen, wird der try-Block beendet und der erste passende catch-Block ausgeführt, sonst werden nach try-Block alle catch-Blöcke übersprungen
    - try-with-resources: try(Klasse id = …; Klasse2 id2 = …; …){…  
      muss Interface AutoCloasable mit void close() implementieren
  + jede Exception muss gefangen oder durch throws-Klausel weitergereicht werden (in den meisten Interpretern bei main keine throws.Klausel erlaubt)  
    Außer java.lang.RuntimeException und Subtypen (sonst Aufwand zu hoch (alle /0, usw.)  
    (Bsp.: ArithmeticException, ClassCastException, IndexOutOfBoundsException, NegativeArraySizeException, NullPointerException (NullPointer auch bei throw null;))
* Exception neben java.lang.Error Subtyp von java.lang.Throwable (letztere aber eher für Entwicklung)  
  Error-Klassen gedacht für Fall, der nicht sinnvoll gefangen werden kann (wird nicht ausversehen durch Exception e gefangen)
* AssertionError extends Error mit verkürzter Schreibweise:  
  assert bedinung: messageOfAssertionErrorAsString;  
  kurz: assert bedinung;   
  (wenn Bedinung falsch wird entsprechender AssertionError geworfen) (Können durch Setzungen für den Compiler abgeschaltet werden)
* Quasi-Standard zum Methodentesten JUnit-Tests (im Gegensatz zu vorherigen Exceptions (White-Box-Test), Black-Box-Test)
  + - Import static org.junit.Assert.assertEquals;
    - Import static org.junit.Assert.assertTrue;
    - Import static org.junit.jupiter.api.Assertions.assertThrows;
    - Import org.junit.jupiter.apiTest;
    - Import org.junit.jupiter.api.BeforeEach;
  + Tests in separate Klasse in dieser:
    - @Test //wird JUnit aufgerufen werden alle Methoden mit dieser Annotierung nacheinander aufgerufen)  
      public void someIdentifier(){/\*zu Testende Anweisungen z.B. assertEquals(2, 1+1);\*/}
    - @BeforeEach //wird direkt vor jeder Testmethode ausgeführt  
      public void name(){/\*Anweisungen\*/}
    - assertEquals(controlWert, wert) liefert fehler, wenn nicht gleich (test mit .equals)
    - assertEquals(controlWert, min, max) wenn nicht in range
    - assertSame (Test auf Objektidentität)
    - assertThrows(Exc.Klassenname.class , ObjektVomInterfacejava.lang.reflect.Executable)  
      (.class Klassenkonstante, Objekt mit Informationen zur Klasse) (funktionale Methode von Executable parameterlose void Methode)

# GUI

## (Applet)

* Eher leichtgewichtig
* Integration in HTML
* <object codebase="link zum Code”

classid="java:applets.Klassenname.class"

codetype="application/x-java-applet”

width="breite” height="höhe”></object>

* entsprechende Klasse wird von java.awt.Applet abgeleitet und die public void paint(Graphics graphics)-Methode überschrieben

## Graphics

* Referenzpunkt für Positionierung von Formen oben links
* Color (new Color(r, g, b) je von 0 bis 255)
* repräsentiert Zeichenfläche und moderiert Zeichnen
* Bsp. Methoden:
  + setColor(Color c)
  + fillOval(x, y, durchmesser1, durchmesser2)
  + drawOval(x, y, d1, d2)
  + draw/fillRect(x,y,width; height)
  + drawLine(startX, startY, endX, endY)

# Mathe

* Bisektionsverfahren zur Nullstellenfindung (Näherung bis zu einer Fehlertoleranz) (Suche durch Einschränkung eines Bereichs mit Vorzeichenwechsel) (in R)

# Random

* Iff steht für „... if, and only if,“
* Prädikat: Funktion mit booleschem Ergebnis