FOP

Table of Contents

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
- Prinzip: Seperation of Concerns (z.B. Thread (Organisation) und Runnable (Inhalt))

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:

    - 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
- Definitionen:
    - Definitionen public oder gar nichts
    - nur eine public pro Datei (muss nicht an erster Stelle sein) (z.B. kein Modifier ist zusätzlich erlaubt)
    - außer nested classes (verschachtelte/eingebettete Klassen)
        - Bsp. public class X{public class Y{}}
        - Begriffe: äußere und innere Klasse
        - innere Klassen können auch private oder protected sein
        - von außen Ansprache der inneren Klasse durch X.Y
        - nicht static:
            - jedes Objekt der inneren Klasse hat einen annonymen Verweis auf ein Objekt dr äußeren Klasse mit/von dem es erstellt wurde und hat auch auf deren private Attribute Zugriff
            - von der äußeren Klasse in Objektmethoden wird implizit this verwendet
            - Konstruktor der inneren Klasse von außen nur mit Objekt der äußeren Klasse: z.B. a.new Y();
        - static inner class
            - Zugriff nur auf Klassenattribute und Klassenmethoden
            - aus anderen Dateien für den Konstruktor nur Angabe der Klasse z.B. new X.Y();
- 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 ... */
- 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)
    - 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);}
    - weder noch: Empfohlen für In/Out-Parameter und Rückgabewert
- 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)
  (equals soll Äquivalenzrelation sein (symmetrie teilweise verletzt, da supertyp subtyp oft als sich selbst erkennt, subtyp den supertyp aber nicht als gleich akzeptiert)
  (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
  (nicht in FOP: seit Java 9 auch private Methoden)
- 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(), Predicate<T>.test(T t)

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))
- double Math.ceil(double d) //aufrunden, Ergebnis muss oft auf int gecastet werden
- java.lang.System
- java.math.BigDecimal beliebig genaue Zahl
- Calendar.getInstance().get(Calender.HOUR_OF_DAY), boolean Calender.before(Calender c)
- new Timestamp(long millisecoundsSinceBegin1970) //timestamp.getHour(), timestamp.getMinute() (java.sql)
- 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, mathematisch strikte schwache Ordnung (alle sortiert, aber auch gleiche))
- Comparable Interface int compareTo
- Arrays.equals(Object[] a, Object[] a2) (auch entsprechende Methode für für primitive Datentypen)
- new Random().nextDouble() (java.util) (auch für int, long und float, bei ersteren aus gesmaten Wertebereich, bei Kommazahlen aus dem Intervall [0, 1)) (Wahrscheinlichkeitsverteilung uniform distribution)

Grundlegende Algorithmen

- Tauschen mit temporärer Zwischenvariable
- Maximumssuche durch Durchlaufen und Zwischenvariable korrigieren („in case of a tie“ first or last)
- Suche nach Nullstelle in Intervall durch Halbieren und Suche nach Vorzeichenwechsel
- (volles Ablaufen eines Raums, Voraussetzung, dass sich Spuren nicht kreuzen alles abdecken und keinen Bereich abschneiden)
- Suche
  - Lineare Suche
  - Binäre Suche ($\log_2(n)$ nur bei sortiertem Array)
- Sortieren
  - Bubblesort (stabil) (Achtung andere Variante als in Schule Element wird von vorne im Zweifelsfall bis ganz nach hinten jeweils paarweise vertauscht, die letzten h (jeden Schleifendurchlauf steigend) Element sind sortiert, erinnert an Insertionsort)
  - Selectionsort (stabil) (immer Suche des letzten max Elements, dann Tausch, danach Reparatur Stabilität: paarweises Durchtauschen nach rechts des nach links getauschten, daher wie weiterschieben aber nur unter gleichwertigen Elementen auf eingeschränktem Bereich)

Javadoc

```
/**
* Beschreibung
*
* @Tag1 ...
* @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
- Streams ähnlich zu Listen, aber potentiell lazily evaluated und nicht in HtDP-TL, aber in Racket
  - (stream-cons x str)
  - (stream-first str) (späteste Moment an dem erstes Element materialisiert werden muss)
  - (stream-rest str) (muss hier noch nicht materialisiert werden)
  - (stream-empty? Str)
  - (stream-map fct str)
  - (stream-filter pred str)
  - (stream-fold init fct str)

7 (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)

7 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

7 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)

7 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)

7 Laufzeitchecks:
- (check-expect wert wert)
- (check-within wert min max)
- (check-error (berechnung) fehler) z.B. „/: division by zero“

7 Konzept Rekursion (Selbstaufruf einer Methode)
- Rekursionsabbruch (sonst Endlosschleife) (und Redunktionsschritt)
- Bsp. Mathe Fibonacci, Pascalsche Dreieck
- Grundlegendes Konzept, da Schleifen funktionalem Konzept widersprechen

7 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?

7 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

7 Laufzeitfehler (nicht von Compiler, Programmabruch, Ausgabe Call-Stack) (z.B. für durch Sprache ausgeschlossene Laufzeitfehler teilen durhc 0 bei double durch *_INFINITY und NaN)

7 Exception Klassen sind alle Subtypen von java.lang.Exception (Konstruktor mit String message and getter) (für debugging exc.printStackTrace();) (häufig wird eigene abgeleitet)

7 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;))

7 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)

7 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)

7 Quasi-Standard zum Methodentesten JUnit-Tests (im Gegensatz zu vorherigen Exceptions (White-Box-Test), Black-Box-Test)

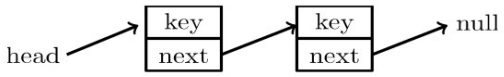
```
Import static org.junit.Assert.assertEquals;
Import static org.junit.Assert.assertTrue;
Import static org.junit.jupiter.api.Assertions.assertThrows;
Import org.junit.jupiter.apiTest;
Import org.junit.jupiter.api.BeforeEach;
```

- 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)
```

Collections

- Interface Collection<T> (java.util)
  - boolean add(T element) throws UnsupportedOperationException (laut offizieller Dokumentation auch erlaubt, dass die Methode immer diese Exception wirft, weil die Dateistruktur kein add zulässt), NullPointerException (manche erlauben kein add(null)), ClassCastException, IllegalArgumentException, IllegalStateException (true, wenn element eingefügt wurde)
  - boolean addAll(Collection<? extends T> collection) throws wie bei add
  - int size()
  - boolean isEmpty()
  - boolean contains(Object element) (verwendet equals) (findet auch null in der Collection falls unterstützt)
  - boolean containsAll(Collection<?> collection)
  - void clear()
  - boolean remove(Object element) (bei mehreren wertgleichen wird eines davon entfernt)
  - boolean removeIf(Predicate<? Super T> pred (gibt true zurück, wenn min. Ein Element entfernt wurde)
  - (keine filer, fold und map Methoden, lassen sich allerdings einfach als Methode die eine Collection übergeben bekommt schreiben)
  - geerbt von Iterable
    - Iterator<T> iterator() (jede Collection gibt ihre spezifische Iterator Klasse zurück, aber wir können einfach Iterator als statischen Typ verwenden)
- Klasse Collections (java.util)
  - static <T> void sort(List<T> list (stabiler Sortieralgorithmus, daher Reihenfolge gleicher Elemente bleibt erhalten), Comparator<? Extends T> comp)
- Interface List<T> (java.util) (erweitert Collection, hat zusätzlich eine Reihenfolge der Elemente))
  Auswahl zusätzlicher Methoden:
  - int indexOf(Object element) (erster Index des Vorkommens, sonst -1)
  - T set(int index, T element) throws IndexOutOfBoundsException, UnsupportedOperationException, NullPointerException, ClassCastException, IlegalArgumentException (Rückgabe ist der alte Wert des Elements)
  - boolean add(T element) (wie Collection nur Garantie, dass das Element ans Ende angehängt wird)
  - void add(int i, T element) (verschiebt weitere Elemente nach hinten)

    - T get(int i)
- Inteface Iterator<T> (java.util) (hat sich auch in anderen Sprachen durchgesetzt)
    - boolean hasNext()
    - T next() (bei List, die Verpflichtung, dass die Reihenfolge dem Index entspricht)
    - default Methode remove() (darf nur einmal nach next() aufgerufen werden)
      Während Durchlauf, darf/soll die Collection sonst eigentlich nicht geändert werden)
    - typische Anwendung:
      while(it.hasNext()){ … it.next(); …}
      Kurzform for(T element : collection){ /*Anweisungen*/ }
- Beispiele Collection-Klassen: Vector, LinkedList, ArrayList, TreeSet, HashSet (keine Eierlegende Wollmilchsau)
- Interface Map<K,V> (java.util)
    - Beispiel: HashMap
    - put(K key, V value) (alle keys in der Map müssen voneinander verschieden sein)
    - V get(K key) (null, wenn key nicht vohanden)
- eigene LinkedList Implementation
    - public class ListItem <T> {public T key; public ListItem<T> next;}
    - Klasse List hat eine Referenz auf ListItem<T> head (zusätzlich z.B. die Länge als Attribut)
    -
    -
    - Für viele 

Methoden Durchgang mit Laufvariable, nennen wir hier statt Laufindex Laufpointer für Vorwärtsversetzung pointer = pointer.next;
      Beispiel: for (ListItem<T> p = head; p != null; p = p.next){}
      Bei einfügen und Löschen umbiegen der Referenzen (wurde in Folien ausführlich implementiert)
      Fall head meist Sonderfall
      Achtung in manchen Sprachen gibt es keinen GarbageCollector
      Iterator hat Laufpointer attribut
      Eigene Iterator Klasse oft als annonyme Klasse
    - Alternatives LinkedList Konzept: jedes Listenelemente hat ein Array von Elementen, dass es verwaltet (ausführliche Diskussion in Folien)
- Doppelt verkettete Listen: Vorteil auch zurückgehen, Nachteil: mehr Speicher und Laufzeit für Änderungen
- Zyklische Listen (letztes Element verweist nicht auf null, sondern wieder auf den head) (Anwendung z.B. in Rundlaufverfahren (Roubd Robin) z.B. Vergabe von Zeit auf dem Prozessor)
- Geafhr mit Listen durch endloses hinzufügen vollaufen des Speichers

Streams und Files

Klasse Optional<T> (java.lang)

- Kapselt ein Objekt oder null ein, bei letzterem wird es leer genannt
- Static Optional<E> ofNullable(E element)
- T get() throws NoSuchElementException (extends RuntimeException, wird geworfen, wenn leer)
- T orElseGet(Supplier<? extends T> other) (Supplier hat eine funktionale Parameterlose Methode mit T als Rückgabe)
- void isPresent(Consumer<? extends T> consumer) (funktionale Methode: void accept(T t)) (nur, wenn nicht leer wird der Consumer ausgeführt)
- Optional<U> map(Function<? super T, ? extends U> mapper)
- Optional<T> filter(Predicate<? super T> predicate) (wenn nicht erfüllt wird ein leeres Optional zurückgeliefert)
- OptionalInt, OptionalLong, OptionalDouble zum direkten Arbeiten mit diesen primitiven Datentypen erstere hat int getAsInt()

Interface Stream<T> (java.util.stream)

- Streams können im Vergleich zu Collections auch unendlich sein
- Erzeugung:

    - jede Liste hat eine list.stream() Methode, die einen entsprechenden Stream vom selben generischen Typ zurückgibt
    - Stream<T> Arrays.stream(T[] array) (java.lang.Arrays)
    - Stream<T> Stream.of(T... values)
    - Stream<String> Files.lines(Path path) throws IOException
- Stream<T> filter(Predicate<? super T> predicate)
- Stream<U> map(Function<? super T, ? extends U> function)
- Optional<T> max(Comparator<T> comparator)
- Iterator<T> iterator()
- T reduce(T identity, BinaryOperator<T> accumulator) //ähnlich zu lfold in racket
  U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)
  `T reduce(BinaryOperator<T> accumulator)`
- T[] toArray(T[]::new) (Number[]::new hier vom Interface IntFunction, bekommt int und liefert Array entsprechender Länge)
- List<T> collect(Collectors.toList()) (Collectors Klasse mit nützlichen Funktion für Streams, keine weiteren Details, Collectors.toList() gibt Objekt vom Interface Collector zurück)
- Zwischenoperstionen (welche, die wieder die gleiche Klasse zurückgeben) nennt man intermediate operations
  <-> terminal operations
- IntStream, LongStream, DoubleStream zum direkten Arbeiten mit primitiven Datentypen in Streams
  z.B. IntStream.of(Int... values)
  IntStream new Random().ints()
  LongStream new Random.longs()
  DoubleStream new Random.doubles() (dazu gibt es auch ein Iterator Interface java.util.PrimitiveIterator.ofDouble (nested class)
- Folien eigener endloser Stream, aber eigentlich nur Iterator Klasse

System (java.lang)

- String getProperty(String key) //wenn kein korrekter key wird null zurückgeliefert (kann SecurityException (Runtime) werfen, wenn kein Zugriff auf das Attribut)
    - „user.home“: Heimatverzeichnis des Nutzers
    - „user.dir“: momentane Arbeitsverzeichnis
    - „user.name“: Benutzername des Nutzers im System
    - „file.seperator“: systemspezifische Trennung zwischen Bestandteilen von Pfaden (fast immer „/“ bei UNIX, „\“ bei Microsoft)
    - „line.seperator“: auch systemspezifisch
- System Bytedatenstreams (auch Standard-Input, Standard-Error, Standard-Output)
    - System.out und System.out PrintStream (standardmäßig Konsole) (Konvention out für normale Ausgeaben, err für Fehler, In der Praxis wird zumindest err meist in eine Log-Datei umgeleitet)
    - System.setOut(PrintStream out), System.setErr(PrintStream err)
    - System.in InputStream (standardmäßig von Tastatur (Konsole?))
    - System.setIn(InputStream inS)
    - (können z.B. auch auf Dateien gesetzt werden)

Path, Paths (java.nio.file)

- Repräsentiert möglichen Dateipfad im System
- Erstellen: Path Paths .get(String first, String... more)

Files (java.nio.file)

- Stream<String> Files.lines(Path path) throws IOException (letztere java.io) //implementiert AutoCloseable (zeilenweises auslesen primär bei Textdateien sinnvoll) Fehler, wenn irgendetwas schiefläuft, z.B. Datei nicht vorhanden
- boolean exists(Path path) //wenn irgendwas im System mit diesem Pfad existiert
- boolean isReadable(Path path) //fragt Zugriffsrechte ab, richtet sich nach Benutzer des Prozess
- boolean isWriteable(Path path) //In typischen Dateisystemen haben Dateien, usw. Dafür attribute

- boolean isRegularFile(Path path) //Datei
- boolean isDirectoy(Path path) //gibt noch ein paar mehr zur Abfrage des Dateityps
- long size(Path path) //Größe in Bytes
- createFile(Path path)
- copy(Path p1, Path p2)
- move(Path p1, Path p2) //auch für rename
- delete(Path p) throws NoSuchFileException
- deleteIfExists(Path p)

Bytedaten (java.io)

- Sinnvoll z.B. für Bilddateien, aber heutzutage für fast alles Standardbibliothek
  Auf dieser Methode baut das textbasierte Lesen auf
- abstract InputStream (implements AutoCloseable) (int read() throws IOException (ließt nächsten byte aus, Rückgabewert int, da negativer Bereich ignoriert, bzw. -1 signalisiert Dateiende)
- FileInputStream (extends InputStream) (new FileInputStream(String filename) throws FileNotFoundException, new(File file))
- abstract OutputStream (implements AutoCloseable) (write(int byte) throws IOException (nur der unterste Byte wird geschrieben, Rest ignoriert)
- FileOutputStream (extends OutputStream) (new(String filename [, boolean append (standard false)]) throws FileNotFoundException (erstellt die Datei, falls noch nicht existent, Fehler z.B. falls directory oder datei nicht erstellbar))
- Da Hardware zumeist auf lesen und schreiben mehrerer Bytes optimiert
  - Puffered Bytes im Puffer und lädt wenn Puffer verwendet in einem Rutsch neue
  - new BufferedInputStream(InputStream in) (extends FilterInputStream extends InputStream)
  - new BufferedOutputStream(OutputStream out) (extends FilterOutputStream extends OutputStream)
- PrintStream (java.io) (extends FilterOutputStream extends OutputStream)
  - sozusagen Konvertierer von String zu Bytes („konfortabler Aufsatz“)
  - new PrintStream(OutputStream out)
  - print(String s) (überladen: auch für primitive Datentypen und Object)
  - println(String s) (wie print nur mit Zeilenumbruch am Ende)
- weitere Klassen von InputStream oder OutputStream abgeleitet
  - java.util.zip.ZipInputStream
  - java.util.jar.JarInputStream (jar sind mit zip komprimierte java Dateien)
  - javax.sound.sampled.AufioInputstream
  - java.io.PipedInputStream
    java.io.PipedOutputStream (erlauben Kopplung miteinander, nur empfohlen bei Threads)
- Textdateien direkt (ohne Streams) (für Text einfachere Methode)
  - Reader reader = new BufferedReader(new FileReader(String fileName)) (beide indirekt Subtypen von java.io.Reader)
    Reader reader = new InputStreamReader(InputStream in) (wenn schon in InputStream offen) (keine Exceptions die gefangen werden müssen)
  - int reader.read(char[] a) throws IOException (schreibt Array voll oder bis Ende der Datenquelle, gibt Anzahl tatsächlich gelesener Bytes zurück)
  - BufferedReader Objektmethode: String readLine() throws IOException (ließt alles vom letzten Zeichen bis nächstes Zeilenende ein, null, wenn Ende der Quelle erreicht))
  - LineNumberReader (new LineNumbeReader(Reader reader) //keine zu fangenden Exceptions, String readLine() throws IOException (null wenn Ender der Quelle erreicht))
  - Writer writer = new BufferedWriter(new FileWriter(String filename))
    Writer writer = new OutputStreamWriter(OutputStream out)
  - write(char c)
  - write(String str)

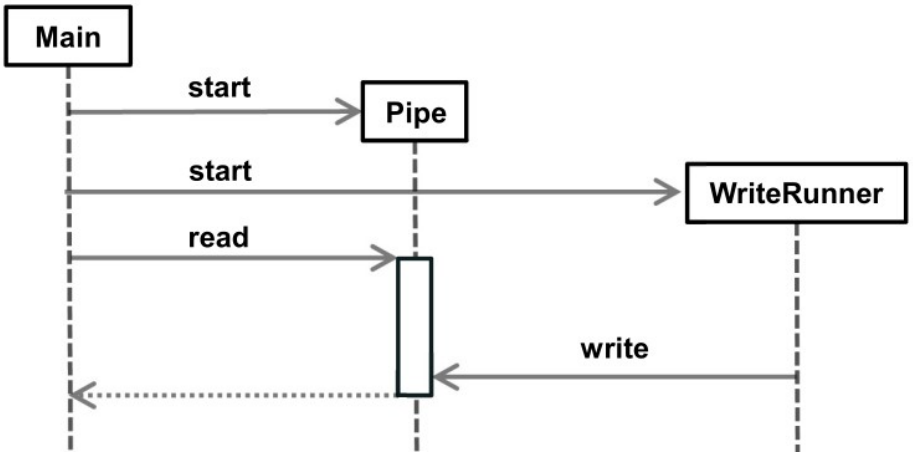
Threads (nicht mehr in FOP)

- Unterscheidung zwischen echten (schwergewichtigen) Prozessen und Threads (leichtgewichtige Prozesse)
  Echte Prozesse werden vom Betriebssystem verwaltet

- eine Programmausführung nur Ausführung eines Threads, daher Unterscheidung einzelner Thread in Programmausführung
- alle Threads laufen (parallel je nach Hardwareplattform bei mehreren Prozessoren potentiell real parallel) unabhängig voneinader bis er seine Arbeit beendet hat (alle anderen laufen dann unbehindert weiter)
- Greifen mehrere Threads auf die gleiche Resource zu kann nicht vorhergesagt werden in welcher Reiehnfolge die Zugriffe sein werden (kann sich auch zwischen Programmläufen unterscheiden)
- Parallelisierung (zur Beschleungigung oder Auteilung auf mehrere Computer) (oft bei numerischen Berechnungne oder Datenbankoperationen)
  - Auteilung meist nur in Anzahl verfügbarere Treads sinnvoll
    int numberOfThreads = Runtime.getRunTime().availableProcessors()-1 (java.lang.Runtime repräsentiert Laufzeitumgebung)) (-1 für den ursprünglichen Thread, der die anderen Threads startet)
  - komplex, daher man muss wissen was man tut, nicht immer schneller
  - Gibt vorgesehene (komplexere) Mechanismen (z.B. Executor, Executors, ExecutorService, Callable, Future, genauer hier nicht teil der FOP)
  - Bsp. Berechnung einer Funktion für alle Elemente eines Arrays
    Aufteilung (Partitionierung) des Arrays in möglichst gleichgroße Segmente, die jeweils ein Thread bearbeitet
    Anschließen wartet der Thread der eingeteilt hat auf die Terminierung aller Threads, indem er immer wieder den Status aller Threads abfragt
- Parallele Streams
  - Stream<T> s = list.parallelStream() //kann muss aber nicht Aufteilung in Threads realisieren
  - bequem und mutmaßlich effizient
- Zweck (Parallelisierung oder) Abspalten von eigenständigen Programmteilen (z.B. GUI)
  - Starten und vergessen („fire and forget“)
  - Starten und später nochmals ansprechen
- von Orcale empfohlene Methode zur Beendigung eines Threads, wenn dieser nicht unverzüglich beendet werden muss, sondern ein regelmäßiges Prüfen des erzeugten Threads, ob er sich beenden soll reicht
  - eigene RunnableKlasse, die von Runnable erbt, mit Attribut toBeTerminated und Methode, die dieses auf true setzt. In run wird regelmäßig (z.B. nach jedem verarbeiteten Wert) gegebenenfalls die run methode beendet.
- Interface Runnable (java.lang)
  - zuständig für Funktionalität eines Threads (seperation of concerns)
  - funktionale methode void run() (Inhalt der Methode ist, was der Thread tut)
- Klasse Thread (java.lang)
  - zuständig für Organisation der Threads
  - new Thread(Runnable runnable) (Referenz zu Thread wird oft auch nicht in Referenz gespeichert)
  - start() (startet eigentlichen Thread) (wirft IllegalThreadStateException (extends RuntimeException) wenn ein zweites mal ausgeführt)
  - static Thread currentThread() //Thread in dem diese Mthode ausgeführt wird
  - dumpStack() //Schreibt Call-Stack in System.err
  - static Map<Thread, StackTraceElement[]> getAllStackTraces() //Call-Stacks aller momentan aktiven Threads, ein Element in StackTraceElement repäsentiert ein Element im Call-Stack
  - long getId() //bleibt konstant und eindeutig in den aktiven Threads (nach Beendung kann die ID neu vergeben werden)
  - String getName() /muss nicht eindeutig sein, entweder durch Konstruktor bestimmt oder gernerisch durch „thread-<number>“ (<number> irgendeine ganze Zahl) gebildet
  - int getPriority(), setPriority(int p) //Bei Java-Programmstart bekommt der erste Thread je nach System und Benutzer einen Prioritätswert (in Klassenkonstante Thread.NORM_PRIORITY). Jeder neue Thread übernimmt diesen Prioritätswert. Änderung im Rahmen der Klassenkonstanten Thread.MIN_PRIORRITY, Thread.MAX_PRIORITY, sonst IllegalArgumentException (extends RuntimeException) falls fehldendes Änderungsrecht SecurityException (extends RuntimeException)
  - static sleep(long millisecounds)
  - Thread.State getState() //Thread.Sate ist ein eingebetteter Enum z.B. Thread.State.TERMINATED
- Pipe

- in Informatik Begriff unidirektionale Datenbrücke zwischen zwei Prozessen, bzw. Threads (einer sendet, der andere empfängt (Eibahnstraße))
- Anzahl von read- und write-Aktionen kann beliebig sein (Daten werden gepuffert bis sie gelsen werden) (werden erst mehrere geschrieben und dann gelesen, bleibt die chronologische Reihenfolge erhalten) (keine mehreren read-Aktionen, da der Thread ja erst auf eine Antwort wartet)
- Erzeugung:
  - PipedOutputStream out = new PipedOutputStream()
    umgekehrt: new PipedOutputStream(PipedInputStream in) throws IOException
    hat print und println Methode
  - new PipedInputStream(out) throws IOException (implements AutoCloseable)
    umgekehrt: new PipedInputStream()
    daraus lässt sich z.B. über einen InputStreamReader ein LineNumberReader erzeugen (diese Umwandlung muss nicht in einem try-catch erfolgen)
  - eines der Objekte kann dann dem neuen Thread gegeben werden
- read (gibt nächsten Inhalt zurück, sobald verfügbar, solange wartet der Thread)

- Sequenzdiagramm (zur Visualisierung)
  - Zeitverlauf nach unten (nur Reihenfolge relevant, keine numerische Skala)
  - Threads beginnen mit Klassennamen in eckigem Kasten (auf der Höhe ihrer Erzeugung) mit gestrichelter Linie nach unten
    links und rechts nebeneinander
    (erster Thread, der mit main-Methdoe)
  - Pipe Thread mit Kasten für einen aktiven Zustand (während eine read-Aktion läuft oder solange eine Nachricht gepuffert wird)
  - Pfeile Nachrichten zwischen den Threads zu bestimmten Zeitpunkt z.B.:
    - start
    - read (zu Pipe) (mit gestricheltem Pfeil der gelesenen Nachricht zurück)S
    - write (zu Pipe)



  - Methodenaufruf z.B. terminate, setVisible(true)

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

Window Manager

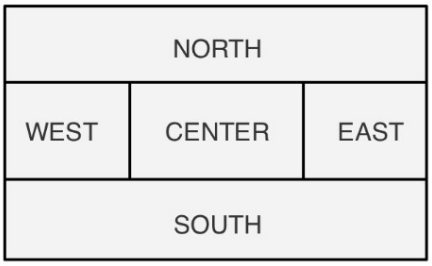
- Systemprozess
  - der in der Regel beim Booten hochgefahren
  - läuft permanent im Hintergrund (service (Windows-Welt), daemon (UNIX-Welt)
- generelle, anwendungsunspezifische Funktionalität
  - Öffnen, Schließen, (De-)ikonoifizieren, Größe ändern
  - Rahmen eines Fensters und Bildschirmhintergrund (Aufbau aus Fenstern und Unterfenstern (auch Desktop Fenster))

Basic Classes

- new Color(int r, int g, int b) //rgb zwischen 0 (nicht) und 255 (voll)
  - Color.RED
- new Font(String schriftfamilie, int schnitt, int size) //z.B. „Helvetica“, Klassenkonstanten BOLD, ITALIC, PLAIN as Bitmask, daher auch ITALIC | BOLD für beides gleichzeitig, z.B. 20

java.awt (abstract window toolkit)

- Component
  - eigener Thread
  - void addFocusListener(FocusListener listener)
  - void addKeyListener(KeyListener listener)
  - void addMouseListener(MouseListener listener)
  - void addMouseMotionListener(MouseMotionListener listener)
  - void addMouseWheelListener(MouseWheelListener listener)
  - void getFontMetrics(Font font)
  - void paint(Graphics graphics) //ist bei allen Componenten zur Darstellung genutzt, wird aber meist nur bei Canvas von einem selbst überschrieben
  - void setBackground
  - void setFont(Font font)
  - void setVisible(boolean visible)
  - Dimension getPrefferedSize()
- Container (extends Component)
  - new Container()
  - void paint() //überschrieben, sodass es auch alle paint-Methoden seiner Components aufruft
  - void add(Component comp) //auch Container und Window oder Frame, …
  - void add(Component comp, Object constraints)
  - void setLayout(LayoutManager manager)
  - void validate() //aktualisiert Layout z.B. nach Änderung einer Font, eines Textes, usw.
    //nicht unbedingt, bevor der Frame visible gesetzt wird?
  - Interface LayoutManager
    - BorderLayout
      - Possible Constraints are Klassenkonstanten vom Typ String: NORTH, WEST, EAST, SOUTH, CENTER (standard is CENTER (in add-Methode ohne Constraint))
      - Aufteilung in fünf Bereiche (Größen von Inhalt abhängig)



    - BoxLayout
      - Komponenten werden in einer Reihe nacheinander ausgelegt (im Konstruktor kann angegeben werden ob horizontal oder vertikal)

        - GridLayout
            - Matrixartig neben- und übereinander (Anzahl Zeilen und Spalten im Konstruktor)
        - FlowLayout
            - Komponenten in ihrer prefferedSize nebeneinader, wenn nicht mehr ausreichend Zeilenumbruch (wie zeilenweises Schreiben bei fester Spaltenbreite)
        - CardLayout
            - Zeigt Komponenten nicht zugleich, sondern nacheinander mit den Methoden first, last, next und previous steuerbar
- Window (extends Container)
    - Fenster ohne Rahmen
    - LayoutManager Standard ist BorderLayout
- Frame (extends Window)
    - Fenster mit Rahmen
    - LayoutManager Standard ist BorderLayout
    - üblicherweise wird eigener Subtyp geschrieben, der Attribute für alle hinzugefügten Komponenten hat
    - Methoden:
        - new Frame(String title) //Title steht später üblicherweise im Rahmen
        - setVisible(boolean visibility) //erst false, damit Fensteraufbau nicht gesehen wird
        - setBackground(Color color)
        - dispose() //Gibt alle Resourcen wieder frei
        - setExtendedState(int state) //Frame.ICONFIFIED, NORMAL, MAXIMIZED_HORIZ
        - add/*ListnerName'/(/*entsprechendes Event*/)
- Komponenten (alle direkt von Component abgeleitet)
    - Button
        - auch eigener Thread
        - new Button(String label) //Text auf dem button
        - setLabel(String label)
        - setFont(Font f)
        - addActionListener(ActionListener actionListener) //registriert ActionListener (auch mehrere)
    - Canvas
        - Hat void paint(Graphics graphics) Methode, die man meist überschriebt (in Canvas leergelassen), denn graphics erlaubt das zeichnen auf die Canvas (graphics mit Zeichenfläche verbunden, alle Zeichanweisungen an graphics werden auf der Zeichenfläche realisiert)
    - Checkbox (verwendet ItemListener)
        - new Checkbox(String label)
        - addItemListener(ItemListener itemListner)
        - boolean isSelected()
        - setLabel(String label)
        - ItemListener (extends java.awt.event) (funktionales Interface)
            - void itemStateChanged(ItemEvent event)
    - Choice (verwendet ItemListener)
        - new Choice()
        - add(String option) //auf Positionen 0, 1, …
        - add(String option, int position) //alle danach rücken nach hinten
        - select(int option)
        - String getSelectedItem()
        - int getSelectedIndex()
        - String getItem(int index)
        - addItemListener(ItemListener itemListener)
    - Label
        - new Label(String label)
        - setAlignment(int Alignment) //Klassenkonstanten: CENTER, LEFT, RIGHT
        - setBackgroundColor(Color c)
        - setText(String label)

- List
  - new List(int numberOfTogetherShownElements, boolean multipleMode) //ist die Anzahl elemente größer, als die gleichzietig gezeigten bietet List eine Scrollbar, MultipleMode erlaubt die Auswahl mehrerer Elemente
  - int[] getSelectedIndexes() //Länge ist Anzahl ausgewählter Elemente, Elementer des Arrays die ausgewählten Positionen
- Scrollbar (verwendet AdjustmentListener)
  - Werden häufig automatisch hinzugefügt
  - new Scrollbar(int orientation, int initialValue, int visibleRange, int min, int max) //Klassenkonstanten: VERTICAL, HORIZONTAL, visibleRange bestimmt die Größe des Balkens zum Scrollen, Range beeinflusst nicht Größe auf Bildschirm, daher kann an Anwendungslogik orientiert sein
  - addAdjustmentListener(AdjustmentListener, adjustmentListener)
  - adjustmentListener (funktionales Interface)
    - void adjustmentValueChanged(AdjustmentEvent event)
    - AdjustmentEvent
      - int getValue()
- TextComponent (verwendet TextListener)
  - Abstract, Abstraktion für zwei direkt abgeleitete Klassen
  - TextField
    - Eine Textzeile
    - new(int numberOfChars)
    - String getText()
    - setEchoChar(char echo) //ersetzt alle chars durch echo (häufig ‚*‘), standard ist kein echo (‚\u0000‘)
    - addKeyListener(KeyListener keyListener)
  - TextArea
    - new TextArea(String initialText, int rows, int columns, int scollbars) //Klassenkonstanten für scrollbar: SCROLLBARS_NONE, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_VERTICAL_ONLY, SCROLLBARS_BOTH
    - addFocusListener(FocusListener focusListener)
    - String getText()
    - setText(String text)
    - FocusListener
      - void focusGained(FocusEvent event)
      - void focusLost(FocusEvent event)

- Listener
  - Event Dispatch Thread, welcher permanent Eingaben mit Maus und Tastatur erwartet und bei jedem Event alle registrierten Listener aufruft
    (vereinfachte Darstellung in Sequenzdiagramm durch vom Laufzeitsystem automatisch eingerichteten dauerhat laufenden Event-Loop Thread)
    z.B. bei Mausklick Berechnung der graphischen Komponente der Koordinaten und Aufruf registrierter Listener
  - abstract Adapter-Klassen, die alle Methoden mit leerem Rumpf implementiert
    (in eigener Klasse müssen sie so nicht extra implementiert werden)
    - KeyAdapter implement KeyListener
    - MouseAdapter implementiert Mouse{ , Motion, Wheel}Listener
    - WindowAdapter implements Window{ , Focus, State}Listener
  - funktionales Interface ActionListener (jawa.awt)
    - bekommt typischerweise Referenz auf Frame oder ist eine innere Klasse des Frames
    - funktionale Methode: void actionPeformed(ActionEvent event)
    - ActionEvent
      - long event.getWhen() //Millisekundan ab Beginn 1970
  - KeyListener KeyEvent

        - void keyPressed(KeyEvent event)
        - void keyReleased(KeyEvent event)
        - void keyTyped(KeyEvent event)
    - KeyEvent
        - int getKeyCode() //dazugehörige Klassnekonstanten KeyEvent.VK_A (Buchstabe A), VK_COLON (Doppelpunkt), VK_BACKSPACE, VK_ENTER, usw.
    - MouseListener MouseEvent
        - void mouseClicked(MouseEvent event)
        - void mousePressed(MouseEvent event)
        - void mouseReleased(Mouseevent event)
        - mouseEntered(MouseEvent)
        - mouseExited(MouseEvent)
    - MouseMotionListener MouseEvent
        - void mouseDragged(MouseEvent event)
        - void mouseMoved(MouseEvent event)
    - MouseEvent
        - int getButton() //Klassenkonstanten BUTTON1, BUTTON2, BUTTON3
        - int getX() //Ursprung ist oben links, daher keine negativen Werte
        - int getY()
    - MouseWheelListener MouseWheelEvent
        - void mouseWheelMoved(MouseWheelEvent event)
    - MouseWheelEvent
        - int getWheelRotation() //number of rotated „clicks“ (bei kleineren Bewegungen 0 bis ganzer Click akkumuliert)
    - WindowListener WindowEvent
        - void windowOpened(WindowEvent event)
        - void windowClosing(WindowEvent event)
        - void windowClosed(WindowEvent)
        - void windowActivated(WindowEvent event)
        - void windowDeactivated(WindowEvent event)
        - void windowIconified(WindowEvent event)
        - void windowDeiconified(WindowEvent event)
    - WindowStateListener WindowEvent
        - void windowStateChanged(WindowEvent event)
    - WindowFocusListener WindowEvent
        - void windowGainedFocus(WindowEvent event)
        - void windowLostFocus(WindowEvent event)

Swing (javax.swing)

- Baut auf java.awt auf, mehr Funktionalität, daher in der Regel bevorzugt
- UIManager (javax.swing)
    - static setLookAndFeel(String name) throws UnsupportedLookAndFeelException, ClassNotFoundException, InstantiationException, IllegalAccessException //Alle Exceptions müssen gefangen werden
    - static getSystemLookAndFeelClassName()
    - //letzteres als Parameter der ersten Methode, setzt das Design auf das Systemdesign
    - für einen Plattformübergreifenden Java-Standard
        - UIManager.setLookAndFeel(LookAndFeel lookAndFeel) throws UnsupportedLookAndFeelException
        - z.B. new MetalLookAndFeel() //plattformübergreifende Java-Standard
        -
- JFrame extends java.awt.Frame
    - eine der Neuheiten: Separierung von Hauptmenü und Rest des Fensters (besonders nützlich auf Plattformen, bei dennen Menü zu Fenster nicht oben im Fenster erscheint? Foliensatz 10 Seite 287)

- JComponent extends jawa.awt.Container //workaround, da keine Mehrfachvererbung wurde JContainer fallen gelassen
- Jcomponent weitere Methoden
  - setToolTipText(String text) //ausschalten mit null (standard), Erzeugt Popup-Fenster, wenn man mit der Maus eine Weile auf dem Element verweilt
  - Randdarstellung:
    - setBorder(Border border)
    - Border BorderFactory.createLineBorder(Color color, int thickness) //rechteckiger Rand mit vier gleiche Linien
    - Border BorderFactory.createBevelBorder(int type) //3D Effekt des Buttons z.B. BevelBorder.LOWERED, .RAISED weitere Konstruktor mit vier zusätzlichen Color Parametern für beteiligt Farben an 3D-Effekt
    - Border BorderFactory.createEtchedBorder(int type) //3D-Effekt bei dem nur Rand höher, niedriger scheint z.B. EtchedBorder.LOWERED, .RAISED
    - Border BorderFactory.emptyBorder() //Zurücksetzen auf Ausgangszustand
  - Vereinfachter KeyListener
    - Hilfsmethode: KeyStroke KeyStroke.getKeyStroke(String keyStroke) //z.B. „alt shift X“
    - getInputMap().put(KeyStroke keyStroke, Object actionMapKey)
    - getActionMap().actionMap.put(Object actionMapKey, Action action) //actionMapKey oft String, aber relevant nur Wertgleichheit
      //value wird bei beiden überschrieben, wenn Schlüssel schon vorhanden, implementieren zwar nicht Map, sind aber ähnlich, wenn value null wird der key Eintrag gelöscht
    - Interface Action (implements java.awt.ActionListener)
      - void actionPerformed(ActionEvent e) //von ActionListener
      - Standartaktionen
        - new StyledEditorKit.UnderlineAction() //Texte werden von jetzt an unterstrichen, bzw. nicht mehr unterstrichen
  - Darg&Drop (Funktionalität geschenkt, Mechanismus selbst änderbar, aber nicht empfohlen)
  - Assistive Technologies (Unterstützungsmöglichkeiten zum Design von Personen mit Handicaps)
- Von JComponent abgeleitet
  - AbstractButton (neu, da man weitere GUI-Komponenten realisieren wollte, sodass sich herausfaktorisieren lohnt)
    - JButton
    - JToggleButton (neu)
      - JCheckBox
      - JRadioButton (neu) (kleiner ancklickbarer Kreis oder Quadrat, meist Teil einer ButtonGroup, in dieser kann nur ein JradioButton angeklickt sein)
    - ...
  - JLabel
  - JList<T> //hier generisch, wenn nicht anders konfiguriert wird toString auf Bildschirm dargestellt (Recherge Einstiegspunkt ListCellRenderer im Package javax.swing)
  - JScrollBar
  - JTextComponent (davon abgeleitet)
    - JTextArea
    - JTextField
    - JFormattedTextField (neu) (Vorformattierung z.B. als Datum, die nur Datumseingaben zulässt, Formatierungsregeln durch java.util.Formatter)
    - JPasswordField (neu)
  - JToolBar (neu) //vereinfachte Möglichkeiten, Standard-Menüs zu arrangieren
  - JSlider (neu) //Schieberegler für Eingaben
  - Popup (neu) //Popup-Fenster
  - JTable (neu) //repräsentiert egwöhnliche Tabelle
    - new JTable(Object[][] entries, Object[] columnNames)
    - setFillsViewportHeight(boolean fillsViewportHeight) //wenn true wird der Table in Höhe gestreckt

um seinen gesamten Platz auszufüllen (z.B. in JScrollPane)
        - int[] getSelectedRows()
        - int[] getSelectedColumns()
    - JScrollPane (neu) //zeigt nur Ausschnitt und bietet horizontale und vertikale Scrollbar
        - new JScrollBar(Component view)

JavaFX

- import javafx…
- Grundlegend neues Package, sehr ausdifferenzierte Klassenhierarchie
- Swing und JavaFX beide in Betrieb
- Einstiegspunkt eines JavaFX-basierten Threads abstract javafx.application.Application (meist eigener Subtyp, in dem man die main-Methode definiert)
- GUI ist eine Stage (javafx.stage.Stage) auf dem es Szenerie gibt
- abstract Application
    - static launch(String[] args) //wirft bei erneutem Aufruf IllegalStateException (Runtime) //wird in der Regel direkt von mainMethode in eigenem Subtyp aufgerufen
    - abstract void start(Stage myStage)
- Stage (javafx.stage)
    - setScene(Scene scene)
    - setTitle(String title)
    - show()
- javafx.scene
    - Scene
        - new Scene(Parent root, double with, double height)
    - Node (extends java.lang.Object) (Vergleichbar zu awt Component)
        - setEventHandler(EventType<T> eventType, EventHandler<? super T> eventHandler)
          //Konvention EventKlasse hat mögliche EventType<T> als Klassenkonstanten
          //Beispiele für EventType<T>: ActionEvent.Action, KeyEvent.KEY_TYPED, KeyEvent.ANY
        - Convenience Methods
            - setOnAction(EventHandler<ActionEvent> eventHandler)
              //wird bei Unterschiedlichen Subklassen unterschidlich ausgelöst
            - setOnKeyTyped(EventHandler<KeyEvent> eventHandler)
            - setOnKeyReleased(EventHandler<KeyEvent> eventHandler)
            - setOnKeyPressed(EventHandler<KeyEvent> eventHandler)
        - Properties z.B. BooleanProperty visible
        - weitere Subtypen von Node
            - Camera (javafx.scene)
            - Canvas (javafx.scene.canvas)
            - ImageView (javafx.scene.image)
            - Shape (javafx.scene.shape)
            - Shape3D (javafx.scene.shape)
            - SwingNode (javafx.embed.swing.SwingNode)
                - Achtung: nicht ganz frei von Problemen
                - new SwingNode()
                - setContent(JComponent content)
                - JComponent getContent()
                - boolean isResizable()
                - resize(double width, double height)
            - …
    - Parent (extends Node) (Vergleichbar awt Container)
        - **protected** ObservableList<Node> getChildren()
        - ObservableList<T> (javafx.collections) (implements java.util.List<T>)
            - boolean add(T t)
            - addAll(T… t)

        - addListener(ListChangeListener<? super T> listner)
          removeListener(ListChangeListener<? super T> listner)
        - ListChangeListener funktionales Interface
          - void onChange(ListChangeListener.Chnage<? extends T> c)
    - weitere Subtypen
        - javafx.scene.Group //Geometrie bestimmt sich nur aus Kindern)
        - javafx.scene.web.WebView //über WebEngine Schnittstelle zu Webseiten
        - ...
- javafx.scene.layout
    - Region (extends Parent)
        - kann neben Kindern weitere graphische Gestaltbarkeit haben
          Breite, Höhe, Farb-, Randgestaltng
        - weitere Subklassen
          - javafx.scene.Control (Subklasse z.B. Button)
          - javafx.scene.chart.Chart (Subklassen z.B. PieChart, XYChart (kartesisches Koordinatensystem)
    - Pane (extends Region) (ähnlich zu awt Container, dass sein eigener LayoutManager ist (Trennung aufgehoben))
        - überschriebt getChildren() public (Subklassen sind dringend gehalten nur eigene Funktionalität hinzuzufügen und nicht die Rückgabe zu ändern)
    - HBox (extends Pane)
        - new HBox()
        - Komponenten werden horizontal nebeneinander platziert, per default von links nach rechts
    - VBox
    - GridPane (extends Pane)
        - new GridPane()
        - setRowIndex(Node node, int row)
        - setColumnsIndex(Node node, int column)
        - "convenience method": add(Node node, int row, int column)
    - BorderPane
        - new BorderPane()
        - setCenter(Node node)
        - setBottom(Node node)
        - setLeft(Node node)
        - setRight(Node node)
        - setTop(Node node)
    - StatusBar
    - ToolBar //kann horizontal oder vertikal sein
    - ImageView
- .control
    - Control (extends javafx.scene.layout.Region)
        - graphische Komponente, die von Endnutzer manipulierbar
        - direkte Subtypen z.B. ButtonBar, MenuBar, ScrollBar, ChoiceBox<T>, TextInputControl
    - Labeled (extends Control)
        - Subtypen z.B. Label, TitledPane
    - ButtonBase (extends Labeled)
        - wie ein Button anklickbar
        - Subtypen z.B. CheckBBox, Hyperlink, MenuButton, ToggleButton
    - Button (extends ButtonBase) (Subtyp von Region (letzte Gemeinsamkeit mit HBox)
        - new Button(String label)
        - setOnAction(EventHandler<ActionEvent> eventHandler) //auch schon in Node, wir bei Button durch Klick ausgelöst
    - TextField (extends TextInputControl)
        - new TextField(String text)

            - setEditable(boolean editable)
        - TextArea (extends TextInputControl)
- Canvas (extends Node)
    - keine Region, da erstmal nur logische Koordinaten für die sich Canvas merkt, was zu zeichnen ist, erst beim Zeichnen auf eine reale Fläche werden diese Koordinaten umgerechnet, die für das Canvas-Objekt reservierte Fläche ist dann eine Region
    - durch EInfügen in Scenerie ergeben sich die Koordinaten auf realen Fläche
    - am besten zu importieren: javafx.scene.*, javafx.scene.paint.*, javafx.canvas.*
    - beginnt links obenm mit 0,0
    - Propertys: z.B: DoubleProperty width, doubleProperty height
    - new Canvas(double width, double height)
    - GraphicsContext getGraphicsContext2D()
    - GraphicsContext (Asulagern des eigentlichen Zeichnens) (ähnlich zu Graphice von awt)
    - setStroke(Paint p)
    - setFill(Paint p) //unterschiedliche Farben für Striche und Füllen
    - fillRect(x, y, width, height)
    - Paint (auch Bitmuster)
        - Color (extends Paint (nicht awt)
        - ImagePattern (extends Paint)
          z.B. new ImagePattern(new Image(inputStream)
    - setTransform(Transform transform)
    - Affine (extends Transform) (javafx.scene.transform.*)
        - affine-lineare Transformationen, daher linear plus Verschiebung
        - üblicherweise scale, rotate, translate (in der Reihenfolge)
        - new Affine() oder new Affine(Transform transform)
        - append(Transform transform)
        - auch convenience methods: appendScale, ...
        - Transform
            - Scale (extends Transform)
                - new Scale(double widthFactor, double heightFactor) //auch Konstruktor mit Wahl eines Ausgangspunkts
            - Rotate (extends Transform)
                - new Rotate(double angle) //angle in Grad Gegenuhrzeigersinn
                - new Rotate(double angle, double pivotX, double pivotY)
            - Translate (extends Transform)
                - new Translate(double xMove, double yMove)
            - Shear (nicht in FOP)
- javafx.event
    - EventHandler<T extends Event>
        - void handle(T event)
    - ActionEvent (Subtyp von Event)

JavaFX: Properties + Bindings

- auch über javaFX hinaus sinnvoll
- Propertys der JavaFX-Klassen werden oft Standard-Properties genannt
- Property-Klasse ähnlich zu Wrapper-Klassen, aber mehr Funktionen
- Interface Property<T> (imports ObservableValue<T>)
    - bind(ObservableValue<T> observable) //ändert sich automatisch mit
- Beispiele:
    - DoubleProperty (auch Subtyp Property<Number>
    - FloatProperty
    - LongProperty
    - BooleanProperty
    - StringProperty

    - ListProperty
- IntegerProperty (Subtyp von ObservableNumberValue) (implements Property<Number>) (javafx.scene.property.IntegerProperty)
    - new IntegerProperty()
    - setValue(Integer value)
    - int get()
    - bind(ObservableValue<Number> observable)
    - NumberBinding add(ObservableNumberValue observable) //NumberBinding Subtyp von ObservableValue<Number>
    - NumberBinding subtract(ObservableNumberValue observable)
    - NumberBinding multiply(ObservableNumberValue observable)
    - NumberBinding divide(ObservableNumberValue observable)
    - Konvention für Zugriff von Public Bereich der Klasse auf IntegeProperty
        - int **final** getName(){ return name.get(); } //im public-Bereich wie normalen getter/setter
        - void **final** setName(int newName){ name.setValue(newName); }
        - IntegerProperty nameProperty(){ return name; } //neu
- Bindings (javafx.beans.binding) (bietet weitere Funktionalität z.B. für NumberBinding, BooleanBinding)
    - static NumberBinding multiply(ObservableNumberValue prop1, ObservableNumbervalue prop2)
    - static NumberBinding add(ObservableNumberValue prop1, ObservableNumbervalue prop2)
    - weitere übliche arithmetische Operationen
    - static ObservableNumberValue Bindings.when(ObservableBooleanValue condition).then(ObservableNumberValue prop1).otherwise(ObeservableNumberValue prop2) //realisiert durch Rückgabe von when von Typ When, mit überladener MEthode then und vier inneren Klassen ConditionBuilder für Number, Boolean, String, sowie ein generischer ConditionBuilder, diese haben dann eine überladene otherwise-Methode
    - static ObservableBooleanValue and(ObservableBoolean value cond1, ObservableBooleanValue cond2)
    - weitere übliche boolean Opertionen z.B. or, not
    - static ObservableBooleanValue greaterThan(ObservableNumberValue prop1, ObservableNumberValue prop2)
    - static ObservableBooleanValue lessThan(ObservableNumberValue prop1, ObservableNumberValue prop2)
    - equal
    - notEqual
    - //fast alle je noch mit entsprechenden primitiven Datentypen (hochgradig überladen)

Graphics (java.awt)

- 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)
    - drawstring(String string, int x, int y) //linke Seite Höhe der Basislinie
    - Rectangle getClibBounds()
        - rectangle.width
        - rectangle.height
        - rectangle.y
        - rectangle.x
    - FontMetrics getFontMetrics()
        - int fontMetrics.getMaxAscent()
        - int fontMetrics.getMaxDescent() //je von Basislinie, daher Summe beider ist Höhe

- int stringWidth(String string)

Korrekte Software

Kriterien

- Kein Programmabbruch durch Fehler (theoretisch auch durch Fehler der Laufzeitumgebung, aber das ist nicht Teil von FOP)
- Termination
  - Wenn Aufgabe erledigt
  - Wenn Befehl zur Termination von außen (für potentiell unendlich lange Prozesse)
- Korrekte Ausgabe und Effekte

Abstraktionsebenen

- Spezifikatorische Ebene
  - Fehler bei umzusetzendem Gedanken
  - z.B. vor 2000 Jahreszahl nur mit zwei Ziffern kodieren, Flugzeuge Schubumkehr ohne Bodenkontakt gesperrt, bei schlechter Witterung Sensorik aber ungenau, von Anwendern Tool zur externen Kommunikation gewünscht, von Entwicklern Tool zur internen Kommunikation entwickelt
- Logische Ebene
  - Umsetzungsfehler (z.b. durch Denkfehler beim Programmieren) (Fehler bei Übertragung der Gedanken in Code)
  - z.B. off-by-one error (im Prinzip richtig, aber eins daneben)
  - kann sich in semantischem Fehler äußern
  - meist schwer zu finden
- Semantische Ebene
  - Wird meist nicht von Compiler gefunden
  - Wirft RuntimeException
  - Was ein gegebenes sprachlich korrektes Programm ausmachttatsächlich macht
  - z.B. int x=0; int y = 1/x; //das wird vermutlich sogar noch von heutigen Compilern gefunden, aber nicht in komplexeren Situationen, daher nach wie vor semantischer Fehler
  -
- Syntaktische Ebene
  - (Grammatik)
  - Findet Compiler
  - Entscheidet, ob Quelltext korrektes Programm der Sprache (Vorausgesetzt lexikalische Ebene korrekt)
  - Kontextfreie Teil wird von formalen Syntaxregeln ausgedrückt (damit die Regeln nicht zu komplex werden oder teilweise eine komplexere Art erforderlich wäre) (daher kein Prüfen auf Kontext)
    - Abweichung in Informatik Einschränkung auf durch formale Regeln festgelegtes
    - Daher in Java passen Typfehler nicht in die Kategorien
  - z.B. korrekte Klammersetzung (genau eine entsprechende schließende, keine Überlappungen)
  - Einhaltung Syntaktische Konstrukte
    - z.B. for(...;...;...), while(...), do ... while(...)
    - java
      - statement ::= <<simple-statement>>; |<<compound-statement>>|<<if-statement>> |<<switch-statement>>|<<while-loop>>|<<do-while-loop>>|<<for-loop>>|<<break-statement>>|<<continue-statement>>|...
      - compound-statement ::= {<<statement-sequence>>}
      - statement-sequence ::= E|<<statement>><<statement-sequence>>
      - if-statement ::= if(<<condition>>)<<statement>>| if(<<condition>>)<<statement>> else <<statement>>
      - switch-statement ::= switch(<<switch-expression>>){<<switch-list>>}
        - switch-list ::= <<case-list>>|<<case-list>><<default-case>>
        - case-list ::= E|<<case-item>><<case-list>>

        - case-item ::= case <<case-expr>>:<<statement-sequence>>
        - default-case ::= default: <<statement-sequence>>
    - while-loop ::= while(<<condition>>)<<statement>>
    - do-while-loop ::= do <<statement>> while (<<condition>>);
    - for-loop ::= <<long-for-loop>>|<<short-for-loop>>
        - long-for-loop ::= for(<<simple-statement>>;<<condition>>;<<expr-statement>>)<<statement>>
        - short-for-loop ::= for(<<type-name>> <<identifier>> : <<lvalue>>) <<statement>>
    - simple-staement ::= <<variable-declaration>>|<<const-declaration>>|<<expr-statement>>
        - variable-statement ::= <<type-name>><<var-list>>
        - var-list ::= <<var-decl>>|<<var-decl>>,<<var-list>>
            - vardecl ::= <<identifier>>=<<expr-statement>>
        -
    - (gut für Detailfragen: z.B. Anweisungsblock kann auch leer sein)
-

- Lexikalische Ebene
    - (Rechtschreibung)
    - Z.B. wile statt while, For statt for
    - Formale Sprachkonstrukt Definition
        - Definiendum ::= Definiens
        - Definiendum (Name des zu definierenden Konstrukts) (muss genau einmal definiert werden)
        - Definiens (der definierende Ausdruck)
            - Verwendet es anderes Konstrukt <<name>>
            - E für leeres Wort
            - | oder
            - Z.B. a…z|A…Z|_|$|<<letter>>
        - Ableitung
            - Rückwaärts ohne Konstruktnamen, wo es mit Epsilon endet kann es sein
            - z.B. c3_PO <- 3_Po <- _Po <- Po <- o <- E
        - java
            - identifier ::= <letter-extended>><<ident-char-list>>
              ident-char-list ::= E | <<ident-char>>|_|$
              letter_extended ::= <letter-extended>> | <<digit>>
              letter ::= a…z|A…Z
              digit ::= 0…9
        - kann direkte und indirekte Rekursion haben

Korrektheit von Klassen

- Darstellungsinvariante (representation invariant) von Klassen und Interfaces
    - Sicht die durch public Attribute und MEthoden vermittelt wird
    - sollte für Nutzer der Klasse sichtbar sein
    - Ein Objekt von Klasse … repräsentiert zu jedem Zeitpunkt seiner Lebenszeit <generelle Beschreibung>; <wichtige Informationen, Bedingungen> (z.B. auch die Inizes sind ab 0 aufsteigend (nicht selbstverständlich)) (z.b. bei List auch kann bei Suche, … RuntimeExcpetion werfen)
- Implementierungsinvariante (implementation invariant) von Klassen)
    - analog für nicht-public Teile
    - Sollte nicht unbedingt für Nutzer sichtbar sein (daher z.B. als Kommentar in der Quelldatei)
    - Attribut … (vom Typ …) hat … (Beschreibung des Aufbaus) (Alle attribute sollten angesprochen werden (sonst sollte es auch aus der Klasse entfernt werden)) (Instruktion wie Klasse zu lesen/(über)schreiben) (beschreibt Logik, daher Ausgangspunkt für Fehlersuche und Weiterentwicklung) (auch erwähnen, wenn etwas konstant ist)

- Bei Vererbung (oder implements)
  - Darstellungsinvariante und protected Teil der Implementierungsinvariante muss übernommen werden
    - erweitert und verfeinert
    - nichts zurücknehmen
    - (Liskov Substitiution Principle (LSP)) (Jede Aussage über das logische Verhalten der Basisklasse muss auch für die abgeleitete Klasse gelten)
    - (private statt protected, macht es sicher zu ändern ohne Subtypen anpassen zu müssen)
  - auf Darstellungsinvariante des Supertypen verweisen und nur neues schrieben, um Redundanzen (und Differenzen) zu vermeiden)

Korrektheit von Subroutinen

  - (Vertrag)
    - Type (nicht bei explizit statischer Typisierung
    - Precondition
    - Returns (traditioneller Unterscheidung)
    - Postcondition (nicht bei referentieller Transparenz (Seiteneffekte)) (implizit auch keine weiteren Effekte)
  - Liskov Substitution Principles
    - Vorbedingungen nur abschwächen
    - Nachbedingungen nur verschärfen (auch keine zusätzlichen Effekte)
  - Vorbedingungen
    - Bei Objektmethoden Implementierungsinvariante (davor eingehalten)
    - Parameter (in Java Typ nicht Teil des Vertrags, weil von Compiler)
    - Variable/Konstante außerhalb der Klasse (public anderer Klassen)
    - Externe Datenquellen
  - Nachbedingungen
    - Objektmethoden -> Implementationsinvariante (danach eingehalten)
    - Rückgabewert
    - Variable außerhalb der Klasse
    - Externe Datensenken
  - Rekursion
    - Rekursionsschritt muss näher an Rekursionsabbruch führen (damit terminiert in endlich vielen Schritten)
    - Hinter Korrektheit Beweisprinzip der Vollständigen Induktion (Induktionsbehauptung (enthält Induktionsparameter, da Induktion für Aussagen über natürliche Zahlen), -anfang, -vorraussetzung (muss auf Anfang zurückführen, -schritt

Korrektheit in Subroutinen

  - Korrektheit von Ausdrücken analog zu Subroutinen
  - Korrektheit bei Verzweigungen, wenn jede Alternative korrekt ist
  - Korrektheit bei Schleifen
    - Schleifeninvariante (z.B. nach h>=0 Schritten ist result==a[0]+a[1]+...+a[h-1], falls n in a[], dann in a[h-1] ... a[a.length]) (nahtloser Übergang in Endergebnis/aussage) (daraus lässt sich Ableiten wie Variablen zu initialisieren) (abstrakte Idee hinter der Schleife)
    - Schleifenvariante (z.B. h erhöht sich um eins, der Intervallbereich wird halbiert)
    - Vollständige Induktion
      - Invariante = Induktionsbehauptung
      - Induktionsanfang: Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor der Schleife erfüllt ist.
      - Induktionsvoraussetzung: Die Invariante gelte für h-1
      - Induktionsschritt: Unter der Voraussetzung sorgt der Body der Schleife dafür, dass die Invariante auch nach h Durchläufen weiterhin gilt.

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