VORSCHAU ZUM ÜBUNGSSKRIPT
EINFÜHRUNG IN DIE INFORMATIK 1

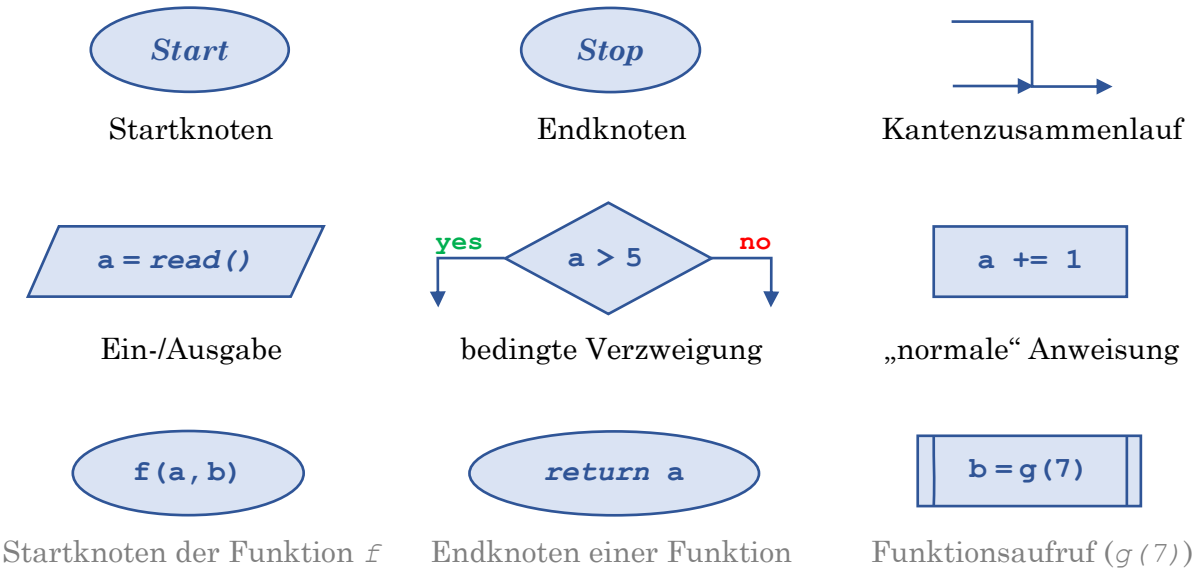
STEFAN BERKTOLD
WWW.STECRZ.DE

VORSCHAU: KAPITEL 10

Kontrollflussdiagramme

10. KONTROLLFLUSSDIAGRAMME

Mit *Kontrollflussdiagrammen* (auch: *Kontrollflussgraphen*) wird veranschaulicht, wie bzw. in welcher Reihenfolge einzelne Programmstücke ausgeführt werden. Es kann bspw. nachvollzogen werden, ob und unter welchen Umständen ein Programm terminiert. In einem solchen Diagramm werden im Grunde alle Statements in Knoten (Ellipse, Rechteck, Raute, Parallelogramm) gezeichnet und mit Kanten (Pfeilen) verbunden. Deklarationen von Variablen (z. B. „`int i;`“) werden dabei nicht eingezeichnet! Es existieren folgende Knotenarten:



Startknoten | Endknoten | Kantenzusammenlauf

Ein-/Ausgabe | bedingte Verzweigung | „normale“ Anweisung

Startknoten der Funktion `f` | Endknoten einer Funktion | Funktionsaufruf (`g(7)`)

Ein Kontrollflussdiagramm verfügt über *genau einen Startknoten* und *beliebig viele Endknoten*. Soll eine ganze Methode/Funktion dargestellt werden, so wird anstelle des normalen Startknotens der *Startknoten für Funktionen* und als Endknoten ein (ggf. leeres) *return*-Statement verwendet. Besteht der Code-Auszug aus mehreren Funktionen, dann existiert für jede Funktion ein separater Startknoten. Der Start-/Endknoten steht in einer Ellipse (wie hier), einem Kreis, oder einem Rechteck mit abgerundeten Ecken.

Grundsätzlich werden alle Anweisungen in Rechtecke geschrieben. Ausnahmen bilden Eingabe- und Ausgabe-Statements wie `read`, `write` oder `System.out.println` (→ Parallelogramm) und bedingte Verzweigungen wie bei `if`, `while` oder `switch-case` (→ Raute).

Sonstige Hinweise:

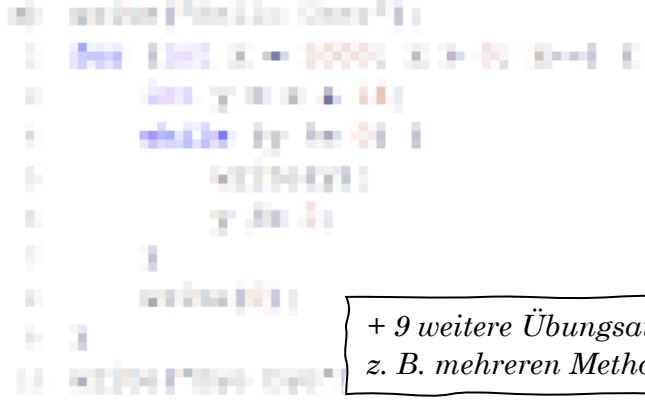
- Das Semikolon (`;`) kann eingezeichnet werden, muss aber nicht. Farben spielen keine Rolle.
- Mehrere aufeinanderfolgende „normale“ Anweisungen (bspw. `a += 1` und `--b`) können in *einem* Knoten zusammengefasst werden (bestenfalls getrennt durch eine horizontale Linie).
- Ob der **`yes`**- bzw. **`no`**-Teil einer bedingten Verzweigung links oder rechts steht, ist egal. Man kann innerhalb eines Kontrollflussdiagramms auch variieren.
- Statt **`yes`** und **`no`** kann auch **`true`** und **`false`**, **`ja`** und **`nein`** oder Ähnliches geschrieben werden.
- Wo die Kanten einen Verzweigungsknoten verlassen oder erreichen spielt keine Rolle.
- **Häufigster Fehler:** Pfeilspitze oder **`yes`**-/**`no`**-Beschriftung vergessen (jeweils -½ Punkt).

㊲ ★★★ Geben Sie die Kontrollflussdiagramme zu folgenden Codeauszügen an.

a)
```
int x = 1;
while (x < 5) {
    int y = x - 1;
    if (y > 3) {
        write("x");
    } else
        break;
    x = x * 2;
}
```

b)
```
int x;
x = read();
while (x < 0 || x > 100) {
    write("Wrong Input");
    x = read();
}
```

c)
```
for (int i = 1; i < 10; i *= 2) {
    if (i % 3 == 0)
        write(i);
}
```



+ 9 weitere Übungsaufgaben mit allen Besonderheiten wie z. B. mehreren Methoden, switch-case, do-while oder Labels

Vorschau: Kapitel 17

Polymorphie

17. POLYMORPHIE

In diesem Kapitel wollen wir uns mit dem Unterschied zwischen statischem und dynamischem Typ sowie deren Bestimmung beschäftigen. Dieses Verständnis ist die Grundlage und Handwerkszeug für Polymorphie. Polymorphie ist ein Konzept der objektorientierten Programmierung, bei dem es im Grunde darum geht, dass eine Variable Objekte unterschiedlichen Typs speichern kann. Schuld daran ist Vererbung (bzw. die damit einhergehende Überschreibung) und Überladung. Eine Methode kann in einer Unterklasse überschrieben werden (gleiche Signatur). Gleichzeitig kann eine Methode überladen sein (gleicher Methodenname, unterschiedliche Parametertypen). Die Signatur der aufgerufenen Methode ist statisch eindeutig bestimmt (durch die statischen Typen), nicht aber die tatsächlich aufgerufene Methode (da ggf. überschrieben)...

I. **Bestimmung des *statischen Typs*** (↝ Typ der Variable):

Der statische Typ einer Variable kann einfach an der *Deklaration* abgelesen werden, d. h. er steht immer links neben dem ersten Vorkommen des Variablennamens (im Gültigkeitsbereich der Variable). Er kann nicht geändert werden (nur „vorübergehend" durch Casten), denn jede Deklaration ist eindeutig.

Sonderfälle: Der stat. Typ von `this` entspricht der Klasse, in der es benutzt wird;
`super` entspricht der Oberklasse der Klasse, in der es benutzt wird.

Besonderheit: *„Casting"* *(vgl. Kap. 2)*

Liegt ein Cast (bspw. `(A)b`) vor, so „ändert" sich der stat. Typ *vorübergehend* (zu `A`).
Wir müssen jedoch zuerst überprüfen, ob der Cast überhaupt funktioniert:

1. *Compiler-Sicht*: Steht die Klasse des *stat. Typs* der Variable, die gecastet wird, in einer Relation (d. h. Vererbungshierarchie) mit der *„Cast"-Klasse*? Wenn die beiden Klassen etwas miteinander zu tun haben, also eine von der anderen erbt (ggf. indirekt) oder die Klassen identisch sind, dann könnte der Cast theoretisch möglich sein. Anderenfalls gibt es einen ***Compiler-Fehler***.
2. *Dynamische Sicht*: Ist das Objekt der Variable, die wir gerade versuchen zu casten, auch tatsächlich vom Typ des „Casts" (oder spezieller)? Wir sehen uns jetzt also den *dyn. Typ* der Variable an (siehe unten) und prüfen, ob diese Klasse (des dyn. Typs) von der „Cast"-Klasse erbt (ggf. indirekt, bzw. identisch sind). Ist der Typ des Objekts nicht mindestens so speziell wie der „Cast-Typ", so tritt ein ***Laufzeitfehler*** (`ClassCastException`) auf.

II. **Bestimmung des *dynamischen Typs*** (↝ Typ des Objekts):

Den tatsächlichen (dyn.) Typ des Objekts, auf das eine Variable verweist, ermittelt man durch *„Zurückverfolgen"* bis zur Objekterzeugung. Wird der Variable `b` bspw. `a` zugewiesen (`b = a;`), so müssen wir nachsehen, auf welches Objekt `a` *zu diesem Zeitpunkt* verweist, um den dyn. Typ von `b` zu bestimmen. Ist die letzte Zuweisung an `a` bspw. `a = new C()`, so zeigt `b` auf ebendieses `C`-Objekt, d. h. `b` den dyn. Typ `C`. Hier interessieren uns also nicht die Deklarationen, sondern die tatsächlichen Zuweisungen der Objekte. Wie die Bezeichnung „dynamisch" schon sagt, ist der dyn. Typ abhängig von der letzten Zuweisung, folglich veränderlich und nur zur Laufzeit des Programms bestimmbar.

III. **Bestimmung der aufgerufenen Methode** / Membervariable für Aufrufe der Form $e_0.method(e_1, \ldots, e_n)$ bzw. $e_0.member$, wobei e_i Ausdrücke (häufig einfach Variablen) sind:

❶ ***Compiler-Sicht*** *(bei allen Methoden oder Attributen)*

1. Bestimme den statischen Typ das Ausdrucks (bzw. der Variable), auf dem die Methode aufgerufen wird (e_0). Eventuell gibt es einen Fehler durch einen Cast.
2. Bestimme – falls nötig – auch die statischen Typen aller Parameter ($e_{1\ldots n}$).
3. Wir suchen nun
   - in der in Schritt 1 gefundenen Klasse (oder einer Oberklasse davon)
   - eine Methode mit passendem Namen (wir nehmen logischerweise nicht `k(...)` wenn wir `m(...)` suchen) und ausreichender Sichtbarkeit, die
   - die Parametertypen aus Schritt 2 akzeptiert. Beachte, dass man einer Methode, die als Parameter z. B. ein `Fahrzeug` erwartet, auch alle Typen (Klassen) übergeben kann, die von `Fahrzeug` erben (bspw. `Fahrrad`).

   Gibt es mehrere passende Methoden (bspw. `m(Fahrzeug)` und `m(Fahrrad)`, wenn wir ein `Mountainbike` übergeben und `Mountainbike` von `Fahrrad` von `Fahrzeug` erbt), so wählen wir immer die bzgl. der Parametertypen speziellste (hier `m(Fahrrad)`). Existiert keine bzgl. *aller* Parameter speziellste Methode (bei mehreren Parametern möglich), so ist der Aufruf *mehrdeutig* → Compiler-Fehler.

   Achtung: Die speziellste Methode könnte auch in der Oberklasse stehen, schließlich erben wir alle nicht-privaten Methoden der Oberklasse! Wir würden im obigen Beispiel also auch dann `m(Fahrrad)` wählen, wenn diese nicht in der aktuellen Klasse aber in der Oberklasse definiert worden wäre. Beachte wiederum, dass wir keinen Zugriff auf private Methoden einer Oberklasse haben.

   Keine passende Methode gefunden bzw. kein Zugriff? → Compiler-Fehler.

❷ ***Dynamische Sicht*** *(nur bei nicht-statischen Methoden)*

Wir führen direkt die in Schritt ❶ gefundene Methode aus, wenn diese `static` (statischer Kontext) oder `final` (sowieso nicht überschreibbar) ist oder wir auf dem Schlüsselwort `super` operieren (also nicht wieder in der Unterklasse kucken wollen). Auch beim Zugriff auf eine Membervariable/Attribut (statt Methode) entfällt der Schritt ❷ (da Variablen wie statische Methoden *nicht* überschrieben werden).

1. Bestimme den dynamischen Typ des Ausdrucks, auf dem die Methode aufgerufen wird (e_0). Sind dynamischer und statischer Typ identisch, so kann man aufhören und die Methode aus Schritt ❶ ausführen. Anderenfalls:
2. Schau nun in der Klasse des dyn. Typ nach, ob die in Schritt ❶ gefundene Methode überschrieben. Dazu muss die Signatur (d. h. der Methodenname und alle Parametertypen) *exakt* übereinstimmen! Ist das der Fall, so führen wir diese Methode aus, anderenfalls die Methode aus Schritt ❶.

   Achtung: Auch hier könnte die überschriebene Methode geerbt worden sein; nämlich von einer Oberklasse des dyn. Typs, die der Klasse des stat. Typs wiederum untergeordnet ist (dyn. Typ ist immer spezieller oder gleich speziell wie stat. Typ).

Wie du siehst ist die Vorgehensbeschreibung ziemlich umfangreich, was insb. den ganzen Sonderfällen geschuldet ist. Daher findest du unterhalb noch **Kurzbeschreibungen** für das Vorgehen bei Polymorphie, welche die grundlegenden Fälle abdecken. Diese Beschreibungen sind im Gegensatz zur vorherigen Erklärung unvollständig, können dir aber beim Einstieg in dieses Thema helfen, falls du auf den vorherigen Seiten den Überblick verloren hast.

Für einen Methodenaufruf der Form **`ausdruck.methode()`**:

1. Bestimme den statischen Typ von **`ausdruck`**, nachfolgend ***StatTyp*** genannt.
2. Suche in der Klasse ***StatTyp*** (bzw. anschließend den Oberklassen von ***StatTyp***) nach einer Methode mit dem Namen **`methode`**, die keine Parameter erwartet. Wird keine passende Methode gefunden, so sprechen wir von einem ***Compiler-Fehler*** und sind fertig.
3. Ist die gefundene Methode statisch (`static`), so bist du fertig und führst die gefundene Methode aus. Anderenfalls bestimme den dynamischen Typ von **`ausdruck`**, also den Typ des Objekts, zu dem **`ausdruck`** auswertet, nachfolgend ***DynTyp*** genannt.
4. Suche in der Klasse ***DynTyp*** (bzw. anschließend den Oberklassen von ***DynTyp***) nach einer Methode mit dem Namen **`methode`**, die keine Parameter erwartet, und führe die gefundene Methode aus. Du findest diese Methode sicher (spätestens die in Schritt 2 gefundene).

Für einen Attributzugriff der Form **`ausdruck.attribut`**:

1. Bestimme den statischen Typ von **`ausdruck`**, nachfolgend ***StatTyp*** genannt.
2. Suche in der Klasse ***StatTyp*** (bzw. anschließend den Oberklassen von ***StatTyp***) nach einer Variable mit dem Namen **`attribut`**. Das Ergebnis ist der Wert dieser Variable.

Für einen Methodenaufruf der Form **`ausdruck0.methode(ausdruck1, ausdruck2, ...)`**:

1. Bestimme den statischen Typ von **`ausdruck0`** und **`ausdruck1`** (und allen weiteren). Diese Typen werden im Folgenden mit ***StatTyp0***, ***StatTyp1***, usw. bezeichnet.
2. Suche in der Klasse ***StatTyp0*** (bzw. danach in den Oberklassen von ***StatTyp0***) nach einer Methode mit dem Namen **`methode`**, die als Parameter die Typen ***StatTyp1***, ***StatTyp2***, usw. „akzeptiert“. Dazu muss die Anzahl der Parameter übereinstimmen. Eine Methode `methode(ParamTyp1 param1, ...)` „akzeptiert“ bspw. ***StatTyp1*** als Parameter genau dann, wenn `ParamTyp1` eine Oberklasse von ***StatTyp1*** ist (oder beide gleich sind). Wird keine passende Methode gefunden, so sprechen wir von einem ***Compiler-Fehler*** und sind fertig. Bei mehreren passenden Methoden: Wähle die speziellste, also die, deren Parametertypen in der Vererbungshierarchie möglichst nah an ***StatTyp1***, ***StatTyp2***, usw. liegen.
3. Ist die gefundene Methode statisch, so bist du fertig und führst die Methode aus. Anderenfalls bestimme den dynamischen Typ von **`ausdruck0`**, nachfolgend ***DynTyp0*** genannt.
4. Ist ***DynTyp0*** gleich zu ***StatTyp0***, so bist du ebenfalls fertig. Anderenfalls suche in der Klasse ***DynTyp0*** (bzw. anschließend den Oberklassen von ***DynTyp0***), ob die zuvor gefundene Methode (aus Schritt 2) hier überschrieben wird. Dazu müssen die Signaturen (Methodenname und Parametertypen) exakt übereinstimmen. Ist dies der Fall, so führe die neue (überschreibende) Methode aus, anderenfalls die in Schritt 2 gefundene.

(89) ★★★ Gegeben seien folgende Klassen (jeweils in einer eigenen Datei):

```
public class F {
    String m = "F.m";
    String m() { return "F.m()" + this.s(); }
    String m(F f) { return "F.m(F)" + s(); }
    private String m(G g) { return "F.m(G)"; }
    String k() { return "F.k()" + this.m(this); }
    String k(G g) { return "F.k(G)"; }
    static String s() { return "F.s()"; }
    static String t() { return "F.t()"; }
}

public class G extends F {
    String m = "G.m";
    final static String s = "G.s";
    String m() { return "G.m()" + super.s(); }
    String m(F f) { return "G.m(F)"; }
    String m(G g) { return "G.m(G)"; }
    String k(F f) { return "G.k(F)" + this.m((G) f); }
    static String s() { return "G.s()"; }
}

public class H extends G {
    String k(F f) { return "H.k(F)"; }
    String k(A x) { return "H.k(A)"; }
    String k(G g) { return "H.k(G)" + super.m((F) g); }
}

public class A extends G {
    F m;
    A(F m) { this.m = m; }
    String k(H t) { return "A.k(H)" + t(); }
    String k(G x) { return "A.k(G)" + x.k(this) + m.m(x); }
}
```

Auf der nächsten Seite finden Sie einige Aufrufe, welche zu `String`s auswerten. Geben Sie für jeden Aufruf an, zu welchem `String` der Aufruf auswertet oder ob der Aufruf zu einem *Compiler-Fehler* bzw. *Laufzeitfehler* führt. Nennen Sie den ggf. auftretenden Fehler. Alle Aufrufe erfolgen aus einer anderen Klasse im selben Package.

Hinweis: Du bist schon geübt im Umgang mit Polymorphie? Schau dir zumindest die Aufrufe ab 57 an!

Folgende Deklarationen und Initialisierungen seien zu Beginn bereits gegeben:

```
F a = new F();
G b = new G();
F c = b;
H h = new H();
A y = new A(h);
```

	Aufruf	Ergebnis
1.	`b.m()`	
2.	`b.m(a)`	
3.	`b.m(b)`	
4.	`b.m(c)`	
5.	`b.k()`	
6.	`b.k(a)`	
7.	`b.k(b)`	
8.	`b.k(c)`	
9.	`b.s()`	
10.	`b.t()`	
11.	`a.m()`	
12.	`a.m(a)`	
13.	`a.m(b)`	
14.	`a.m(c)`	
15.	`a.k()`	
16.	`a.k(a)`	
17.	`a.k(b)`	
18.	`a.k(c)`	
19.	`a.s()`	
20.	`a.t()`	
21.	`c.m()`	
22.	`c.m(a)`	
23.	`c.m(b)`	
24.	`c.m(c)`	
25.	`c.k()`	
26.	`c.k(a)`	
27.	`c.k(b)`	
28.	`c.k(c)`	
29.	`c.s()`	
30.	`c.t()`	
31.	`a.m`	
32.	`b.m`	
33.	`c.m`	

	Aufruf	Ergebnis
34.	`((F)b).m((F)a)`	
35.	`((G)c).m((F)b)`	
36.	`((F)c).m(c)`	
37.	`((F)b).k((F)b)`	
38.	`((G)a).k(c)`	
39.	`((G)c).k((G)c)`	
40.	`h.k()`	
41.	`h.k(c)`	
42.	`h.k(b)`	
43.	`h.k(g)`	
44.	`((F)h).k(b)`	
45.	`((F)h).k(c)`	
46.	`((G)h).k(a)`	
47.	`((G)h).k(b)`	
48.	`((G)h).k(c)`	
49.	`((G)h).k((G)c)`	
50.	`h.k(h)`	
51.	`h.s()`	
	`a = h; // gilt ab jetzt`	
52.	`((H)a).k(b)`	
53.	`((H)a).m(h)`	
54.	`((G)a).k(h)`	
55.	`a.k(a)`	
56.	`a.m(h)`	
57.	`A.t()`	
58.	`H.s`	
59.	`A.k(h)`	
60.	`y.k(h)`	
61.	`new H().k(y)`	
62.	`((H)y).k(c)`	
63.	`y.k(b)`	
64.	`y.k(y)`	
65.	`((G)y).k(h)`	

Die **Lösung** zur vorherigen Aufgabe findest du unter *stecrz.de/files/skript/Loesungen2020-4.pdf*

Weitere Übungsaufgaben findest du im Skript (*shop.stecrz.de/produkt/skript*)

und in EIDI- bzw. PGdP-Altklausuren (*shop.stecrz.de/altklausuren*)

Polymorphie EXTREM

Aufgrund der Steigerung des Schwierigkeitsgrades der Polymorphie-Klausuraufgabe über die letzten Jahre soll diese Aufgabe bereits vorab einen Nervenzusammenbruch garantieren, um diesen während der Klausur zu vermeiden.

Du solltest zuvor die anderen Polymorphie-Aufgaben aus dem Skript bearbeitet haben und die einfacheren Altklausuraufgaben (d. h. bis inkl. WS 16/17) gut lösen können, bzw. dich sehr sicher bei Polymorphie fühlen. Diese Aufgabe wird dir selbst mit perfektem Wissen über Polymorphie noch offene Wissenslücken aufzeigen. Solltest du diese Aufgabe fehlerfrei lösen können, dann ... niemand kann diese Aufgabe fehlerfrei lösen. Solltest du diese Aufgabe aber zumindest größtenteils lösen können, dann bist du wirklich gut auf Polymorphie vorbereitet. Die Lösungen sind getestet (auch wenn du es an mancher Stelle nicht wahrhaben wollen wirst).

In dieser Aufgabe ist neben den für Polymorphie typischen Kenntnissen auch das Verständnis folgender Java-Konzepte notwendig:

- *Generics, generische Vererbung und Type Erasure (Übersetzen von Generics)*
- *Überladung und Mehrdeutigkeit bei Methoden und Konstruktoren (ambiguity)*
- *Ketten polymorpher Zugriffe (`a().b().c()`)*
- *primitive Datentypen, implizites Typecasting, Wrapper-Klassen, Autoboxing (→ Kap. 9)*
- *Sichtbarkeitsmodifier `private` (→ wird nicht vererbt, aber ggf. über `super` zugreifbar)*
- *Ausdrücke und Operatoren (Bindungsstärken), insb. Zuweisungen der Form:*
  `(a = b).m()` (→ stat. Typ von „`(a = b)`“ ≙ stat. Typ von `a`; dyn. Typ ≙ dyn. Typ von `b`)

Die nachfolgende Aufgabenstellung bezieht sich auf die Klasse `Poly`, welche du auf den folgenden Seiten findest. `Poly` enthält mehrere statische innere Klassen. Statische innere Klassen sind im Gegensatz zu nicht-statischen inneren Klassen nicht an eine Instanz der umliegenden Klasse gebunden. `Poly` stellt die Methode `out` bereit. Diese erwartet beliebig viele Objekte (mind. eines) als Parameter und gibt diese in der Form `o1(o2, o3, ...)` auf der Konsole aus. Beachte, dass jede Art von Zahlen (außer `char`) in Java bekanntlich stets dezimal und Gleitkommazahlen immer in Punktnotation ausgegeben werden. Beispiele:

- *`out("f")` gibt `f()` aus*
- *`out("g", 1)` gibt `g(1)` aus*
- *`out("F.m", 'a', 1.1, 0x20, .1)` gibt `F.m(a,1.1,32,0.1)` aus*

Ich empfehle dir (insb. bei Teilaufgabe 3), verschiedene Skizzen – z. B. ähnlich zu einem Objektdiagramm - anzufertigen, um den Überblick zu behalten. Die Lösung enthält eine solche Skizze.

100 ★★☆ **Polymorphie mit allem (und scharf)**: Diese Aufgabe ist in mehrere unabhängige Teilaufgaben gegliedert. Geben Sie jeweils für jeden der markierten Aufrufe an, welche Ausgabe dieser produziert. Sollte ein Aufruf Ihrer Meinung nach zu einem Compiler- oder Laufzeitfehler führen, so ist mit Begründung anzugeben, wo und weshalb dieser auftritt. Außerdem sind alle bis zum Auftreten einer Exception produzierten Ausgaben trotzdem anzugeben, bspw. in der Form: „`Y.m() Z.k(14)` - *dann ArithmeticException (division by zero)*“. Gehen Sie für die übrigen Aufrufe davon aus, dass die nicht funktionierenden Aufrufe auskommentiert sind.

```
public class Poly {
 public static void out(Object m, Object... o) {
   System.out.println(m + "(" + java.util.Arrays.stream(o)
        .map(Object::toString)
        .collect(java.util.stream.Collectors.joining(",")) + ')');
 }
 static class T<Gen> {
   public Gen t;
   public T(Gen t) { out("T","Gen"); this.t = t; }
   public T(Integer x) { out("T","Integer"); this.t = null; }
   public void m(short x) { out("T.m","(short)"+x);
                            this.m(x - 0b10); m((int)x, (int)x); };
   public void m(double x) { out("T.m",x); };
   public void m(double x, int y) { out("T.m",x,y); }
   private void m(S<Gen> s) { out("T.m",s); m((T<Gen>)s); }
   public void m(T<Gen> t) { out("T.m","T"); }
   public T<Gen> m() { out("T.m"); return this instanceof S ?
                                         ((S<Gen>)this).t : this; }
 }
 static class U extends T<Character> {
   public U(char c) { super(c); out("U","char"); }
   public U() { super('A' + 1); out("U"); }
   public void m(int x) { out("U.m",x); this.m(x * .1, x); }
   public void m(int x, double y) { out("U.m",x,y); }
   void m(S<Character> s) { out("U.m","S"); super.m(s); s.m(t); }
 }
 static class X<Z> extends S<Z> {
   public X(Integer i) { super(i); out("X","Integer");
                         t = (T<Z>) new U(); }
   public X(Object o, T<Z> t) { super(o); out("X","Object","T");
                                this.t = t; }
   public void m(int x, int y) { out("X.m",x,y); };
   public void m(int x) { out("X.m",x); super.m(x, x); }
   public void m(double x) { out("X.m",x); };
   public void m(T<Z> t) { out("X.m","T"); }
   public void m(S<Z> s) { out("X.m", "S"); super.m(s);
                           this.t.m(s); s.m(this); }
   public void m(U u) { out("X.m", "U"); m(u.t); t.m().m(u.t); }
 }
 static class S<P> extends T<P> {
   protected T<P> t;
   public S() { this(null); out("S"); }
   public S(int i) { super(i); out("S","int"); t = new X<P>(i, this);
   public S(Object o) { super((P)o); out("S","Object"); }
   public void m(double x, int y) { out("S.m",x,y); m(y); }
   public void m(U u) { out("S.m","U"); this.m(u.t); t.m(this); }
   public T<P> m() { out("S.m"); this.m(this); return super.m(); }
   public String toString() { return "S"; }
 }
```

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

        {    // Teilaufgabe (1) – Primitve Datentypen & Ambiguous Calls

             T<Integer> t1 = new S<>();
             X<Integer> x1 = new X<>(null, null);
             T<Integer> t2 = x1;
             U u1 = new U();
             T<Character> t3, t4 = u1;
             S<Integer> s1;

             byte b1 = 1, b2 = 2;
             char c = 'A';
             short sh1 = 3, sh2 = 4;
             int in = 5;
             float f = sh1 * 2;
/* Aufruf */
    /* 1: */   t1.m(c);
    /* 2: */   t1.m(b1);
    /* 3: */   t1.m(b1 + 1);
    /* 4: */   t1.m(2, 4);
    /* 5: */   t1.m(sh2 - sh1);
    /* 6: */   ((X<Integer>) t1).m(in);
    /* 7: */   ((S<Integer>) t1).m((long)in);

    /* 8: */   t2.m(b2);
    /* 9: */   t2.m(in);
   /* 10: */   t2.m(in == 1);
   /* 11: */   t2.m(in, in);
   /* 12: */   t2.m(sh2, b1);

   /* 13: */   x1.m(5/2);
   /* 14: */   x1.m(b1);
   /* 15: */   x1.m(1l, 07);
   /* 16: */   x1.m(sh1, in);
   /* 17: */   ((S<Integer>) t2).m(0x9);
   /* 18: */   ((S<Object>) t2).m(b1);
   /* 19: */   ((S<Integer>) t2).m(2, 4);
   /* 20: */   ((X<Integer>) t2).m(c);
   /* 21: */   ((S<Integer>) t2).m((byte)f);
   /* 22: */   ((U) t2).m(in);
   /* 23: */   (t3 = t2).m(in);
   /* 24: */   (s1 = x1).m(in, in);
   /* 25: */   (t3 = x1).m(in);
   /* 26: */   ((T<Integer>)(s1 = x1)).m(in, new Integer(9));

   /* 27: */   u1.m(b1 + sh1);
   /* 28: */   u1.m((int) 2*f);
   /* 29: */   u1.m(sh1, in);
   /* 30: */   u1.m(3/2, 1L);
   /* 31: */   u1.m(f, f);
   /* 32: */   u1.m(f, b1);
   /* 33: */   t4.m(in, f);
   /* 34: */   t4.m(in, in);
   /* 35: */   ((S<Character>) u1).m(in);
   /* 36: */   ((S<Character>) t4).m(5d);
   /* 37: */   (t3 = u1).m(in);
   /* 38: */   u1.m(in = (short)sh1);
   /* 39: */   t4.m(13, f = (int)in);
        }  // end: innerer Block zu Teilaufgabe 1
```

VORSCHAU: KAPITEL 19

Lambdas & Streams

19. Lambdas & Streams

Streams kommen aus der funktionalen Programmierung. Ein ***Stream*** repräsentiert eine geordnete Sequenz von Elementen, ähnlich zu einer Liste. Während eine Liste aber eine „echte" Datenstruktur ist, die die *konkreten* Daten (i. d. R. Referenzen auf Objekte) speichert und uns deren direkte Veränderung erlaubt, ist ein Stream eher als eine *abstrakte* Kapselung von Daten zu sehen. Streams erlauben uns, mit nur wenig Code (ohne aufwendige Schleifen) viele Operationen hintereinander auf alle Daten anzuwenden. Der Stream selbst wird dabei allerdings eigentlich nie *echt* verändert. Wenden wir eine der zahlreichen Stream-Methoden an, die die Daten des Streams *scheinbar* verändert (sog. *intermediäre* Operationen), wird tatsächlich ein *neuer* Stream erzeugt, der nun einfach andere Daten repräsentiert. Auf diesem neuen Stream können wir nun wiederum eine Stream-Operation ausführen, bspw. um den Stream weiter zu „verändern". Dies erlaubt uns die Verkettung mehrerer „verändernder" Methoden durch Punktnotation. Man kann sich sozusagen Daten formen. Zuletzt möchte man die im Stream gekapselten Daten jedoch zumeist verwenden/ausgeben/modifizieren. Hierfür gibt es *abschließende* Stream-Operationen.

Beispiel:

Gegeben sei ein Array `names`, das die Namen von Nutzern speichert, wobei die Namen nicht in korrekter Rechtschreibung eingegeben wurden. Wir möchten nun die Namen aller Nutzer, deren Name mit „C" beginnt, in alphabetischer Reihenfolge und korrekt geschrieben (d. h. erster Buchstabe groß, Rest klein) ausgeben. Die ursprünglichen Daten (Array `names`) sollen dabei jedoch nicht verändert werden, da sie später noch benötigt werden.

```
String[] names = {"norbert", "CORNELIUS", "cOrDuLa", "BEN", "", "Carmen"};
```

Lösung ohne Streams:

```
LinkedList<String> modifiedNames = new LinkedList<>();

for (String name : names) { // iteriere über das names-Array
    if (!name.equals(""))
        name = Character.toUpperCase(name.charAt(0)) // erster Buchstabe groß
               + name.substring(1).toLowerCase();    // Rest klein

    if (name.startsWith("C"))
        modifiedNames.add(name);   // Zwischenspeichern ist für das Sortieren nötig!
}

java.util.Collections.sort(modifiedNames); // alphabetisch sortieren

for (String name : modifiedNames) // verbleibende Elemente einzeln ausgeben:
    System.out.println(name);     // Ausgabe: Carmen, Cordula, Cornelius
```

Lösung mit Streams (werden in diesem Kapitel erläutert):

```
java.util.Arrays.stream(names)                    // erzeuge Stream, der die ...
  .map((name) -> name.equals("") ? ""             // ... names-Elemente kapselt
                 : Character.toUpperCase(name.charAt(0))   // erster B. groß
                   + name.substring(1).toLowerCase())      // Rest klein
  .filter(name -> name.startsWith("C"))           // nur Vornamen mit C
  .sorted()                                       // alphabetisch sortieren
  .forEach(System.out::println);                  // Elemente einzeln ausgeben
```

19.1. *Exkurs*: Anonyme Klassen & Lambda-Ausdrücke (->)

Manche Methoden erwarten als Parameter ein Objekt eines Interface-Typs. Wie wir wissen, können Interfaces jedoch nicht instanziiert werden. An solchen Stellen übergeben wir dann typischerweise eine Instanz einer Klasse, welche dieses Interface implementiert. Ein Beispiel dafür ist der Konstruktor der Klasse `Thread`, welcher ein `Runnable` (Interface) erwartet. Dummerweise müssen wir nun extra eine eigene Klasse schreiben, die die Methoden des Interfaces implementiert, oder? Ja, das geht, aber in so einer Situation können wir auch eine *anonyme Klasse* verwenden (siehe Beispiele in Kap. 21!), d. h. wir geben wie sonst auch in geschweiften Klammern eine Implementierung für unsere „Klasse" an, die das entsprechende Interface implementiert. Anstelle des üblichen „`public class MyClass`" setzen wir unmittelbar vor den Klassenrumpf jedoch eine Instanziierung mit dem Namen des jeweiligen Interfaces. Die Klasse trägt keinen Namen, es existiert aber eine Instanz von ihr – man nennt sie daher *anonym*.

Ist das gegebene Interface außerdem *funktional* (d. h. es definiert nur *eine* abstrakte Methode), so können wir die Instanziierung noch weiter abkürzen. Wir schreiben:

```
(nameParam1, nameParam2, ...) -> { /* Methodenrumpf */ }
```

Diese Form heißt *Lambda*-Ausdruck. Die Parametertypen entfallen, da sie eindeutig aus der einzigen Methodensignatur im Interface hervorgehen. Der Methodenrumpf kann beliebigen Code enthalten. Fordert die Signatur der Methode jedoch eine Rückgabe (ist also nicht `void`), so ist wie üblich ein `return`-Statement nötig. Besteht der Methodenrumpf wiederum nur aus dem `return`-Statement, so kann man auf das `return`-Schlüsselwort und die geschweiften Klammern verzichten. Bei einem einzelnen Parameter sind außerdem die runden Klammern nicht nötig.

1. **Allgemeines Beispiel** für Lambdas:

```
interface Function {
    int perform(int a, int b);
}

Function adder = (a, b) -> a + b;
Function divider = (x, y) -> {
    if (y == 0) return -1;   // division by zero
    return x / y;
};

System.out.println(adder.perform(2, 6));   // 8
System.out.println(divider.perform(7, 2)); // 3
```

Dieser Lambda-Ausdruck entspricht folgender Objekterzeugung mittels anonymer Klasse:

```
Function adder = new Function() {
   public int perform(int a, int b) {
      return a + b;
   }
};
```

2. **Konkretes Beispiel** `Stream.iterate(0, x -> x+2)` aus Kapitel 19.2:

   Der Lambda-Ausdruck „`x -> x+2`" entspricht: `(x) -> { return x+2; }`
   `Stream.iterate` erwartet als zweiten Parameter einen `UnaryOperator<T>`, was ein funktionales Interface mit der abstrakten Methode `T apply(T t)` ist. Der Lambda-Ausdruck ist also eine Kurzschreibweise für Instanziierung durch folgende anonyme Klasse:

```
Stream.iterate(0, new UnaryOperator<Integer>() {
    public Integer apply(Integer x) {
        return x+2;
    }
})
```

19.2. Erzeugen eines Streams

Nicht in dieser Vorschau enthalten

19.3. Methodenreferenzen

Nicht in dieser Vorschau enthalten

19.4. Stream-Operationen

Nicht in dieser Vorschau enthalten

Dieses Beispiel-Array sei für die Code-Beispiele in den folgenden Unterkapiteln gegeben:

```
Spieler[] mannschaft = {
    new Spieler("Max", 2),
    new Spieler("Luis", 3),
    new Spieler("Ed", 2),
    new Spieler("Tom", 1),
};
```

```
public class Spieler {
    public int tore;  public String name;
    public Spieler(String n, int t) {
        name = n; tore = t;
    }
    public String toString() {
        return name + "[" + tore + "]";
    }
}
```

19.4.1. Abschließende Methoden (für einen `Stream<T>`):

`long count()` gibt die Länge des Streams zurück (und beendet ihn).

`void forEach(`*`myConsumer`*`)`
mit `myConsumer` ist Funktion `void accept(T t)`

Wendet die übergebene Funktion *`myConsumer`* auf jedes Element des Streams an.

1. Beispiel: Gibt die Strings des Arrays in Großschreibung aus.

```
String[] lang = {"de", "En", "es"};
Arrays.stream(lang).forEach(s -> System.out.println(s.toUpperCase()));
```

2. Beispiel: Setzt die Anzahl an Toren aller Spieler, die 2 Tore haben, auf 0 (hier für Max und Ed).

```
Arrays.stream(mannschaft).forEach(spieler -> {
    if (spieler.tore == 2)
        spieler.tore = 0;
});
```

`boolean anyMatch(`*`myPredicate`*`)` *analog:* **`allMatch`**, **`noneMatch`**
mit `myPredicate` ist Funktion `boolean test(T t)`

Gibt **`true`** zurück, wenn die Funktion *`myPredicate`* für mind. ein Element des Streams wahr ist (also zu **`true`** auswertet), **`false`** sonst. (Kurzschlussauswertung)

1. Beispiel: Prüft, ob *mind. ein* Spieler mehr als 3 Tore geschossen hat (hier nicht, also *keine* Ausgabe).

```
if (Arrays.stream(mannschaft).anyMatch(spieler -> spieler.tore > 3))
    System.out.println("Es gibt einen Spieler mit über 3 Toren");
```

2. Beispiel: Prüft, ob *alle* Elemente aus 4, -6, 2, 8 gerade sind (ja, also folgt die Ausgabe).

```
if (IntStream.of(4, -6, 2, 8).allMatch(x -> x % 2 == 0))
    System.out.println("Alle Elemente sind gerade.");
```

`A[] toArray(`*`arrayConstructor`*`)`
mit `arrayConstructor` ist z. B. `Spieler[]::new` *(bei primitiven Streams kein `arrayConstructor`)*

Packt die Elemente des Streams in ein neues Array des übergebenen Typs, wobei dieser dem Typ aller Elemente (bzw. einem Obertyp) entsprechen muss. Ein Stream von `Spieler`-Objekten kann bspw. nicht in ein `String`-Array gepackt werden.

1. Beispiel: Packt die Elemente des Streams (1, 6, 3) in ein Integer-Array (`int[]` wäre nicht möglich!).

```
Integer[] arr = Stream.of(1,6,3).toArray(Integer[]::new);
System.out.println(Arrays.toString(arr));  // Ausgabe: [1, 6, 3]
```

2. Beispiel: Erzeugt ein `String`-Array aus den Spielernamen (also `["Max","Luis","Ed","Tom"]`)

```
String[] names = Arrays.stream(mannschaft).map(s->s.name).toArray(String[]::new);
```

3. Beispiel: Erzeugt ein `int`-Array aus den Toren der Spieler (Vorgriff auf primitive Streams).

```
int[] spielerTore = Arrays.stream(mannschaft).mapToInt(s -> s.tore).toArray();
```

Erklärungen zu den anderen abschließenden
und intermediären Methoden für Streams (und
primitive Streams) sowie Stream-Aufgaben
findest du im Skript (*shop.stecrz.de/produkt/skript*).

(112) ★★☆

Lösen Sie folgende Aufgaben ohne die Verwendung von Schleifen oder Rekursion:

1. Schreiben Sie eine Methode `int[] primes(int max)`, welche ein Array zurückgibt, das alle Primzahlen beginnend bei 2 bis höchstens `max` (inkl.) enthält.

   Hinweis: Eine Primzahl ist eine ganze Zahl größer als 1, die nur durch 1 und sich selbst teilbar ist.

2. Schreiben Sie eine Methode `long[] morePrimes(int count)`, welche ein Array zurückgibt, das die ersten `count` vielen Primzahlen enthält.

3. Schreiben Sie eine Methode `void fibonacci(int n)`, welche die ersten `n` Zahlen der Fibonacci-Folge kommagetrennt ausgibt. Die erste Zahl der Fibonacci-Folge sei 1.

   Beispiel: `fibonacci(7)` gibt „`1, 1, 2, 3, 5, 8, 13`“ aus.

   Tipp: Verwenden Sie den Ansatz `….iterate(new long[] {0, 1}, arr -> …)…`

(113) ★★★

Ergänzen Sie folgende Methode an den gekennzeichneten Stellen, sodass diese die als String übergebene Hexadezimalzahl als Dezimalzahl zurückgibt (angelehnt an Aufgabe 31). Gültige Hexadezimalzahlen bestehen lediglich aus den Ziffern `0` bis `9` und den Kleinbuchstaben `a` bis `f` und beginnen mit `0x`. Wird ein ungültiger Wert für `hex` übergeben (z. B. `0xA`), so soll eine `IllegalArgumentException` geworfen werden.

Beispiel: *`hexToDez`*`(0x13b)` gibt 315 (= $\mathbf{1} * 16^2 + \mathbf{3} * 16^1 + \mathbf{11}$) zurück.

Hinweis: `map` und `reduce`…

```
public static long hexToDez(String hex) {
    if (hex == null || !hex.startsWith("0x"))
        ┌──────────────────────────────────────┐
        │                                      │
        └──────────────────────────────────────┘

    return hex.chars()   // IntStream über alle Zeichen von hex
        ┌──────────────────────────────────────┐
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        │                                      │
        └──────────────────────────────────────┘;
}
```