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 $e_0.method(e_1, \dots, e_n)$ 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 ($e_{1\dots n}$).
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.

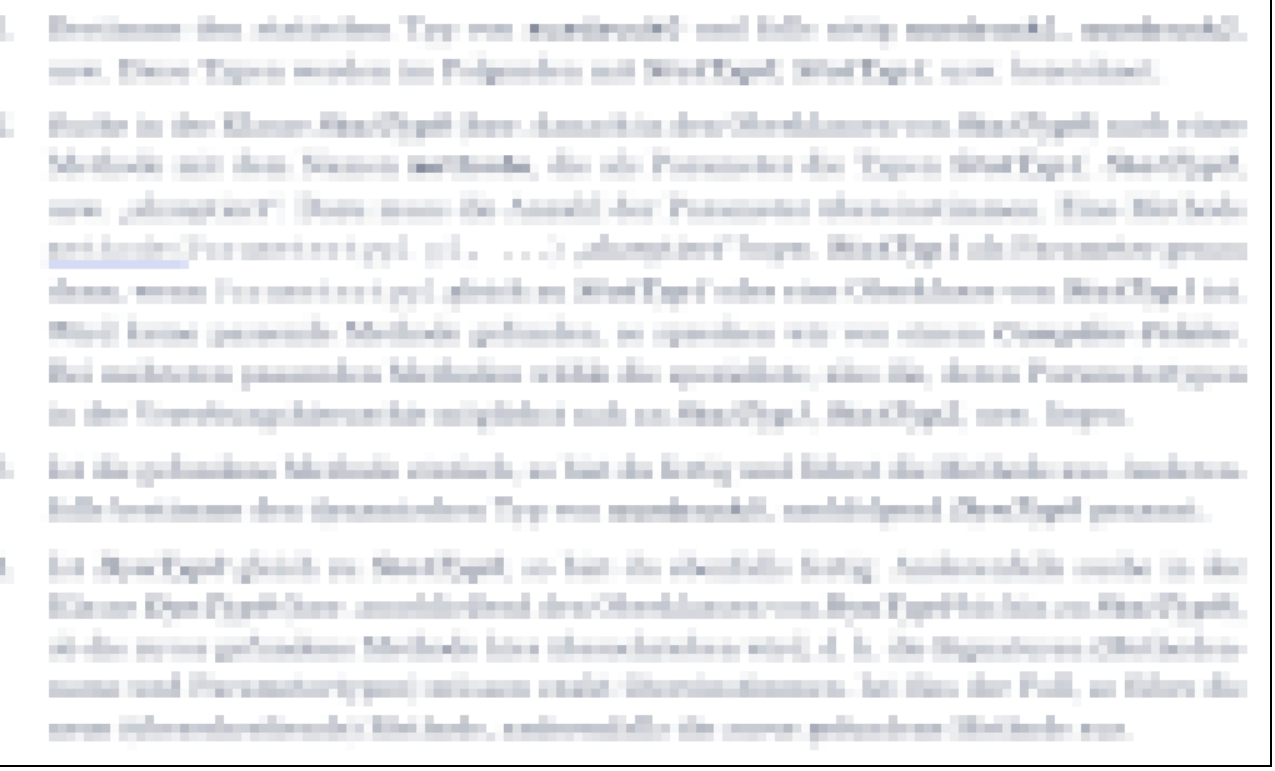
Für einen Methodenaufruf der Form **`ausdruck.methode()`**:

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***.
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.

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, ...)`**:



87 ★★★ 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/Loesungen2020-2.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. 3)*
- *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.

(95) ★★☆ **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
```