# 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"***

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 (dynamischen) Typ einer Variable 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. Ist die letzte Zuweisung an `a` bspw. wiederum `a = new B()`, so zeigt `b` also auf genau dieses `B`-Objekt, daher hätte dann `b` den dyn. Typ `B`. Hier interessieren uns also nicht die Deklarationen, sondern die tatsächlichen Zuweisungen der Objekte. Wie der Name „dynamisch" schon sagt ist der dyn. Typ abhängig von der letzten Zuweisung, d. h. er ist veränderlich und nur zur Laufzeit des Programms bestimmbar (man muss es „durchspielen").

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

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

1. Bestimme den statischen Typ (wie in I. beschrieben) der Variable, auf der 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 `Fahrrad` erwartet, auch alle Typen (Klassen) übergeben kann, die von `Fahrrad` erben (bspw. `Mountainbike`).

   Gibt es mehrere passende Methoden (bspw. `m(Object)`, `m(Fahrrad)` und `m(Mountainbike)`, wenn wir ein `Mountainbike` übergeben und `Mountainbike` von `Fahrrad` erbt), so wählen wir die speziellste (hier `m(Mountainbike)`). Wo ein `Object` als Parameter erwartet wird kann man übrigens alles übergeben, schließlich erbt jede Klasse von `Object`.

   Achtung: Die speziellste Methode könnte auch in der Oberklasse stehen, schließlich erben wir alle Methoden der Oberklasse! Wir würden im obigen Beispiel also auch dann `m(Mountainbike)` 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? → 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 (wie in II. beschrieben) der Variable, 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 in Schritt 1 gefundenen Klasse (dyn. Typ) nach, ob die gefundene Methode aus Schritt ❶ überschrieben wird. Dazu muss die Signatur (d. h. der Name der Methode 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 übergeordneten Klasse, die der Klasse des stat. Typs wiederum untergeordnet ist („Zwischenklassen“).

(85) ★★★ 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 {
    private String m = "H.m";
    String k(F f) { return "H.k(F)"; }
    String k(G g) { return "H.k(G)" + super.m((F) g); }
}
```

Geben Sie an, welche der folgenden (nächste Seite) Statements nicht ausführbar sind (d. h. zu einem *Compiler-Fehler* führen) und welche zu einem *Laufzeitfehler* (Exception) führen. Nennen Sie eine ggf. auftretende `Exception` namentlich. Für alle funktionierenden Aufrufe ist anzugeben, wozu diese auswerten. Anführungszeichen können weggelassen werden. Alle Aufrufe erfolgen aus einer anderen Klasse im selben Package.

Folgende Deklarationen und Initialisierungen seien zu Beginn bereits gegeben:

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

Vervollständigen Sie zunächst folgende Tabelle:

| Variable: | a | b | c | h |
|---|---|---|---|---|
| statischer Typ: | | | | |
| dynamischer Typ: | | | | |

| | Statement | 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;` | |

| | Statement | 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(a);` | |
| 42. | `h.k(b);` | |
| 43. | `h.k(c);` | |
| 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).k(a);` | |
| 54. | `((H)a).m(h);` | |
| 55. | `((G)a).k(h);` | |
| 56. | `a.k(a);` | |
| 57. | `a.m(a);` | |
| 58. | `a.m(h);` | |
| 59. | `b.m;` | |
| 60. | `c.m;` | |
| 61. | `c.s();` | |
| 62. | `F.m;` | |
| 63. | `G.t();` | |
| 64. | `H.s;` | |
| 65. | `b.s;` | |