Zusammenfassung: Statischer Typ, dynamischer Typ & Methoden-Polymorphie

Dieses Kapitel erklärt, wie Java entscheidet, welche Methode tatsächlich ausgeführt wird, wenn wir mit Vererbung und polymorphen Variablen arbeiten. Es geht darum, wie Variablen deklariert sind `statischer Typ` und welches `Objekt` sie zur Laufzeit tatsächlich enthalten `dynamischer Typ`.

Diese Unterscheidung ist zentral für Polymorphie.

1. Statischer und dynamischer Typ

- ✔ `Statischer Typ`
  - Der Typ, mit dem eine Variable im Code deklariert wird.
  - Bestimmt, welche Methoden überhaupt aufgerufen werden dürfen (Compiler entscheidet).

Beispiel:

```
Tier t = new Hund();
```

Tier ist hier der statische Typ.

- ✔ `Dynamischer Typ`
  - Der Typ des Objekts, das die Variable zur Laufzeit tatsächlich enthält.
  - Bestimmt, welche Methode schlussendlich ausgeführt wird.

Im Beispiel oben: Hund ist der `dynamische Typ`.

2. Methodenüberschreiben `Overriding`

Eine `Subklasse` kann eine `Methode` der `Superklasse` neu definieren.

Beispiel:

```
class Tier {
    public void geraeusch() {
        System.out.println("Tier macht ein Geräusch");
    }
}

class Hund extends Tier {
    @Override
    public void geraeusch() {
    System.out.println("Wuff!");
    }
}
```

Jetzt gilt:

```
Tier t = new Hund();
t.geraeusch(); // Ausgabe: Wuff!
```

Der `statische Typ` (Tier) sagt nur: „du darfst `geraeusch()` aufrufen“. Der `dynamische Typ` (Hund) bestimmt: „du sollst die Hund-Version ausführen“.

◇ 3. Methoden-Polymorphie

Polymorphie bedeutet:

- derselbe Methodenaufruf führt je nach `dynamischem Typ` zu unterschiedlichem Verhalten.
- Das macht Programme flexibel, erweiterbar und elegant.

◇ 4. Methodensuche & Methodenauswahl

Zuerst entscheidet der Compiler anhand des `statischen Typs`, ob der Aufruf erlaubt ist.

Dann entscheidet Java zur Laufzeit anhand des `dynamischen Typs`, welche überschreibende Methode aufgerufen wird.

Das Ganze wird „dynamisches Binden“ genannt.

◇ 5. Aufruf der Superklassen-Methode: super

Wenn eine `Subklasse` eine `Methode` überschreibt, kann sie trotzdem die `Superklasse-Version` aufrufen:

Beispiel:

```
class Hund extends Tier {
    @Override
    public void geraeusch() {
        super.geraeusch(); // ruft Tier-Version auf
        System.out.println("Wuff!");
    }
}
```

6. protected

Elemente mit `protected` sind:

- sichtbar innerhalb der Klasse,
- in `Subklassen` (auch in anderen Paketen),
- aber nicht für alle anderen Klassen.
- Gut für kontrollierte Vererbung.

7. toString überschreiben

Alle Klassen erben `toString()` von Object. Um aussagekräftige Ausgaben zu erhalten, überschreibt man diese Methode.

Beispiel:

```
class Auto {
private String modell;

    @Override
    public String toString() {
        return "Auto: " + modell;
    }
}
```

Praktisches Beispiel: Statischer & dynamischer Typ in Aktion

```
class Tier {
    public void geraeusch() {
        System.out.println("Ein Tier macht ein Geräusch");
    }
}

class Katze extends Tier {
    @Override
    public void geraeusch() {
        System.out.println("Miau");
    }
}

public class Demo {
    public static void main(String[] args) {
        Tier t1 = new Tier();
        Tier t2 = new Katze();  // polymorph

        t1.geraeusch();  // Ein Tier macht ein Geräusch
        t2.geraeusch();  // Miau (dynamischer Typ entscheidet!)
    }
}
```

Merksätze

`Statischer Typ` = im Code deklarierter Typ einer Variable.

`Dynamischer Typ` = tatsächliches Objekt zur Laufzeit.

Überschreiben macht `Subklassen` flexibel und ermöglicht Polymorphie.

Methodenaufrufe werden zur `Laufzeit` basierend auf dem `dynamischen Typ` ausgewählt.

`super` ruft die `Methode` der `Superklasse` auf.

`protected` erlaubt Zugriff in der Vererbung, aber nicht von überall.

`toString()` sollte überschrieben werden, um hilfreiche Objektbeschreibungen zu liefern.