Polymorphie

Eine kurze Übersicht für das Vorgehen bei Polymorphieaufgaben.

1: Dynamischer und Statischer Typ

Dynamischer Typ

Der dynamische Typ wird mit der Instanziierung eines Objektes (also mit *new*) festgelegt.

```
Object o = new Fahrzeug();   // dynamischer Typ: Fahrzeug
```

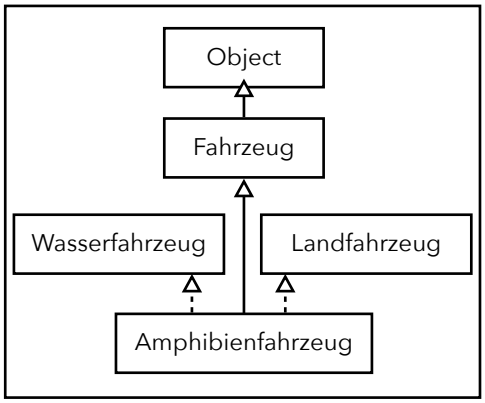
Statischer Typ

Der statischer Typ wird durch den Typ des Attributs bestimmt und kann durch Casts (entsprechend der unten angegebenen Regeln) verändert werden.

```
Object o = new Fahrzeug();   // statischer Typ von o: Object
Fahrzeug f = (Fahrzeug) o;   // statischer Typ von f: Fahrzeug
Amphibienfahrzeug a = (Amphibienfahrzeug) o;   // throws ClassCastException
```

2: Erlaubte Casts

Komplexe Datentypen



Instanzen komplexer Datentypen dürfen in alle statischen Datentypen gecastet werden, die in der Vererbungshierarchie (inklusive Interfaces) über ihrem dynamischen Datentyp liegen. Der dynamische Datentyp ist durch die Instanziierung mit *new* bestimmt und kann nicht verändert werden (man kann dem Attribut aber einen neuen Wert zuweisen).
Für das angegebe Vererbungsschema sind folgende Casts möglich:

```
Amphibienfahrzeug af = new Ambphibienfahrzeug();
(Wasserfahrzeug) af;   (Landfahrzeug) af;          (Fahrzeug) af;
(Object) af;           (Amphibienfahrzeug) af;
```

Casts, die gegen die Hierachie verstoßen, sind nicht möglich. Es wird eine *ClassCastException* geworfen. Beispiele hierfür sind:

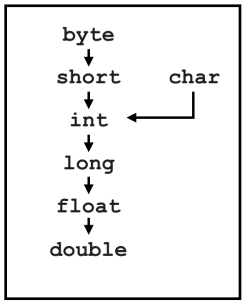
```
Fahrzeug f = new Fahrzeug();
(Wasserfahrzeug) f;      // throws ClassCastException
(Amphibienfahrzeug) f;   // throws ClassCastException
```

Das (dynamische) *Fahrzeug* dürfte allerdings in ein *Object* gecastet werden, da *Object* in der Vererbungshierarchie über *Fahrzeug* liegt.

```
(Object) f;
```

Es wird implizit in Pfeilrichtung gecastet. Casts entgegen der Pfeilrichtung müssen eplizit angegeben werden!

Primitive Datentypen



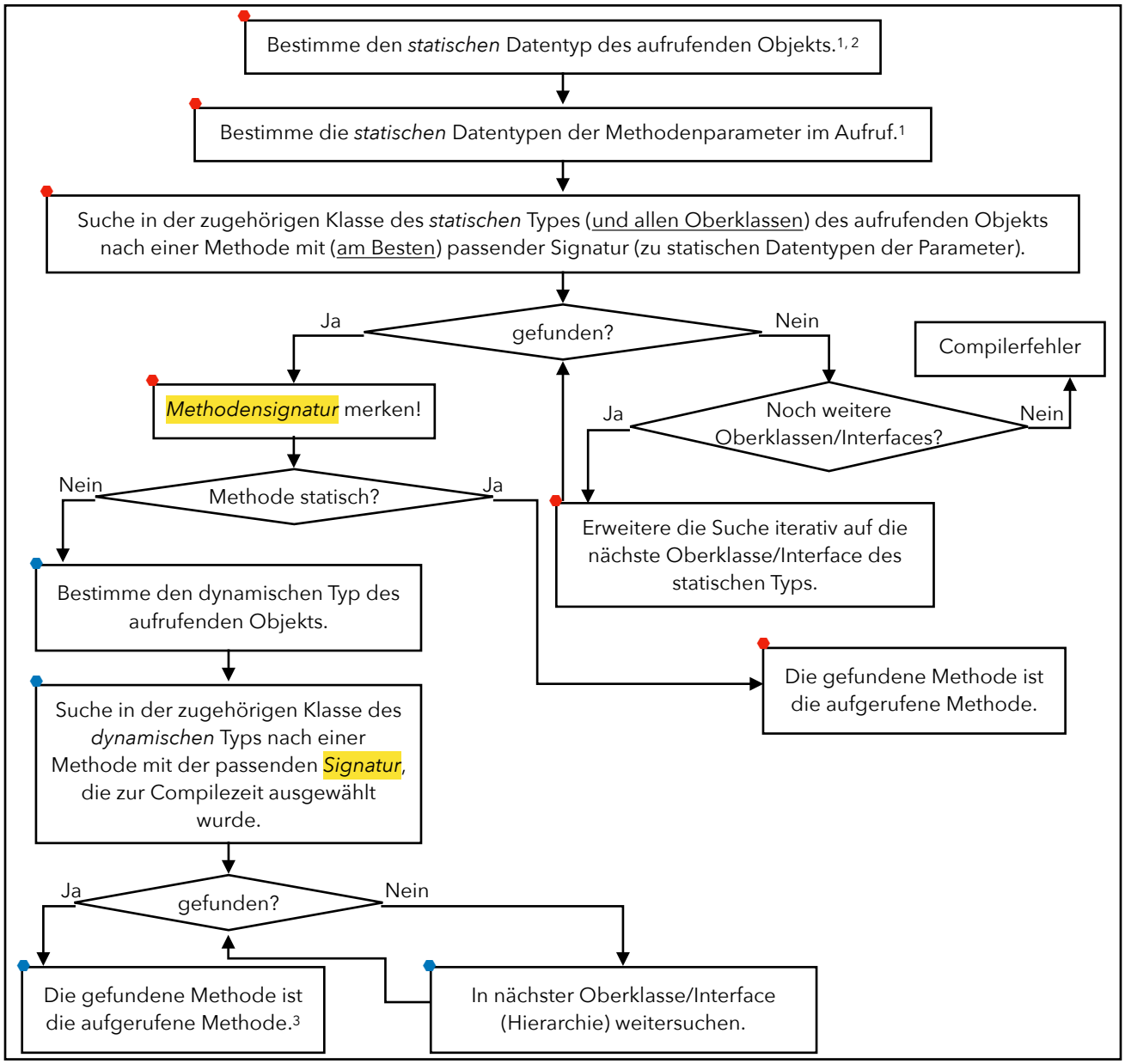
Bei primitiven Datentypen gilt das Schema aus der Übung. Es wird implizit in Pfeilrichtung gecastet. Casts entgegen der Pfeilrichtung müssen explizit angegeben werden.

```
double d = 5;          // impliziter Cast
byte b  = (byte) 162; // expliziter Cast
byte b2 = 162;        // Compilerfehler
```

3: Aufgerufene Methoden bestimmen

Grundsätzlich lässt sich die Auswahl der Methode in zwei Phasen unterteilen.

1. Compilezeit: Während das Programm kompiliert wird, sind ausschließlich die statischen Typen bekannt. Entsprechend wird in dieser Phase nur mit statischen Typen gearbeitet.
2. Laufzeit: Während das Programm läuft, sind auch die dynamischen Typen bekannt. In der Phase wird die finale Methodenauswahl mit der Kenntnis über den dynamischen Typ durchgeführt.



---

1 Bei *this* entspricht der statische Typ der Klasse, in der sich das *this* befindet. Für *super* ist die entsprechende Oberklasse der zugehörige statische Typ.

2 Interfaces und abstrakte Klassen verhalten sich wie normale Oberklassen.

3 Ist die zur Laufzeit (Phase 1) ausgewählte Methode eine andere als die zur Compilezeit (Phase 2) ausgewählte Methode, spricht man von einem *Dynamic Dispatch*.

4: „Best fitting“ Method(e)

Passende Methode in der Klasse des statischen Typs

Die am besten passende Methode (zur Compilezeit) ist diejenige, die mit den wenigsten impliziten Casts (in Pfeilrichtung (siehe Abbildung oben)) auskommt. Am einfachsten findet man die passende Signatur (zur Methode), wenn es bereits eine Signatur für die statischen Typen der Aufrufparameter gibt.

```
class Fahrzeug { void g(Fahrzeug fz) { write(1); } }  // Klasse Fahrzeug
Fahrzeug f = new Amphibienfahrzeug();
f.g(f);  // Aufruf ist statisch: Fahrzeug.g(Fahrzeug), Methode direkt gefunden
```

Passende Methode in den Oberklassen des statischen Typs

Die Suche beschränkt sich nicht nur auf die Klasse des statischen Typen sondern wird bei erfolgloser Suche iterativ weiter nach oben (zu den Oberklassen) ausgeweitet.

```
class Fahrzeug /* extends Object (implizit) */ { }  // Klasse Fahrzeug
class Object { String toString() { return "ein Objekt"; } }  // Klasse Object
 Fahrzeug f = new Amphibienfahrzeug();
 f.toString();  // Aufruf ist statisch: Fahrzeug.toString()
                // Methode nicht in der Klasse des statischen Types
                // Suche in nächster Oberklasse (Object) forsetzen (usw.)
```

Passende Methode mit Casts (implizit)

Es kann allerdings auch sein, dass Casts nötig werden, um die statischen Typen der Parameter einer Methodensignatur anzugleichen. Das Kriterium für die am besten passende Methode ist dann die Anzahl der benötigten Cast (möglichst wenig Casts). Man zählt dabei jeden Pfeil im Vererbungsschema als einen Cast.

```
class Fahrzeug {
  void g(Fahrzeug fz) { write(1); }
  void g(Object o) { write(2); }
}  //Klasse Fahrzeug
Fahrzeug f = new Amphibienfahrzeug();
Amphibienfahrzeug af = new Amphienbienfahrzeug();
f.g(af);  // Aufruf ist statisch: Fahrzeug.g(Amphibienfahrzeug)
  /*
   * Für g(Fahrzeug) ist 1 Casts nötig
   * Für g(Object) sind 2 Casts nötig
   * > g(Fahrzeug) ist die besser passende Methode (weniger Casts)
   *   > Signatur wird ausgewählt (speziellste Signatur)
   */
```

5: Fehler

Compilezeit (Phase 1)

Zur Compilezeit gibt es immer dann einen Fehler, wenn für den statischen Typen des aufrufenden Objekts keine Methode mit passender Signatur gefunden werden kann. In einer IDE (z.B. Eclipse) wird euch dieser Fehler direkt (rot unterstrichen) angezeigt.

```
interface Landfahrzeug  {
  default void g() { write(3); }
}  // Interface Landfahrzeug
Fahrzeug f1 = new Fahrzeug();  Fahrzeug f2 = new Amphibienfahrzeug();
f1.g();  // es gibt in und oberhalb von Fahrzeug keine Methode Fahrzeug.g()
f2.g();  // > Compilerfehler
```

Ein weiterer Fehler, der zur Compilezeit auftreten kann, ist ein „Ambiguous Call“. Dieser Fehler tritt auf, wenn sich der Compiler nicht für eine Methode entscheiden kann, da mehrere Methoden gleich gut passen.

```
  class A extends B { void f(Object o, String s) {} }
  class B { void f(String s, Object o) {} }
  (new A()).f("", "");  // statische Signatur ist A.f(String, String)
```

Der Compiler kann sich hier nicht entscheiden, welche Methode am Besten passt, da für beide Aufrufe ein impliziter Cast (String zu Object) nötig wäre.

Laufzeit (Phase 2)

Zur Laufzeit können Fehler auftreten, wenn illegale Casts durchgeführt werden. Beispiele dafür sind auch bei *erlaubte Casts* aufgelistet.

```
 class Fahrzeug {
   void g(Fahrzeug fz) { ((Landfahrzeug) fz).g(); }
 }  //Klasse Fahrzeug
interface Landfahrzeug  {
   default void g() { write(3); }
 }  // Interface Landfahrzeug
 Fahrzeug f = new Fahrzeug();
 Fahrzeug af = new Amphienbienfahrzeug();
 f.g(af);  // Aufruf wird durchgeführt, da der Cast möglich ist
 f.g(f);   // throws ClassCastException, da f in der Methode g nicht zum
           // Landfahrzeug gecastet werden kann (siehe Vererbungshierarchie)
```

Ein weiterer möglicher Fehler zur Laufzeit ist die NullPointerException. Dieser Fehler wird geworfen, wenn eine Methode auf eine nicht-initialisierte Variable (bzw. den auf `null` zeigenden Pointer) aufgerufen wird. Eine Variable wird durch das Wort **new** und den Konstruktoraufruf initialisiert. Auch bei Parametern muss man darauf achten.

```
 class Fahrzeug {
   void pr(Object o) { write(o.toString()); }
   void foo() { write(5); }
 } //Klasse Fahrzeug
 Fahrzeug f1, f2;        // Variablen f1 & f2 erstellen
 f1 = new Fahrzeug();    // Variablen f1 initialisiert
 f1.foo();               // funktioniert
 f2.foo();               // throws NullPointerException
 f1.pr(f2);              // throws NullPointerException, da toString() in pr nicht
                         // aufgerufen werden kann - äquivalent zu f1.pr(null);
```

Du hast einen Fehler gefunden?
Schreibe mir bitte eine Mail, damit ich das korrigieren kann.
tobias.schamel@tum.de