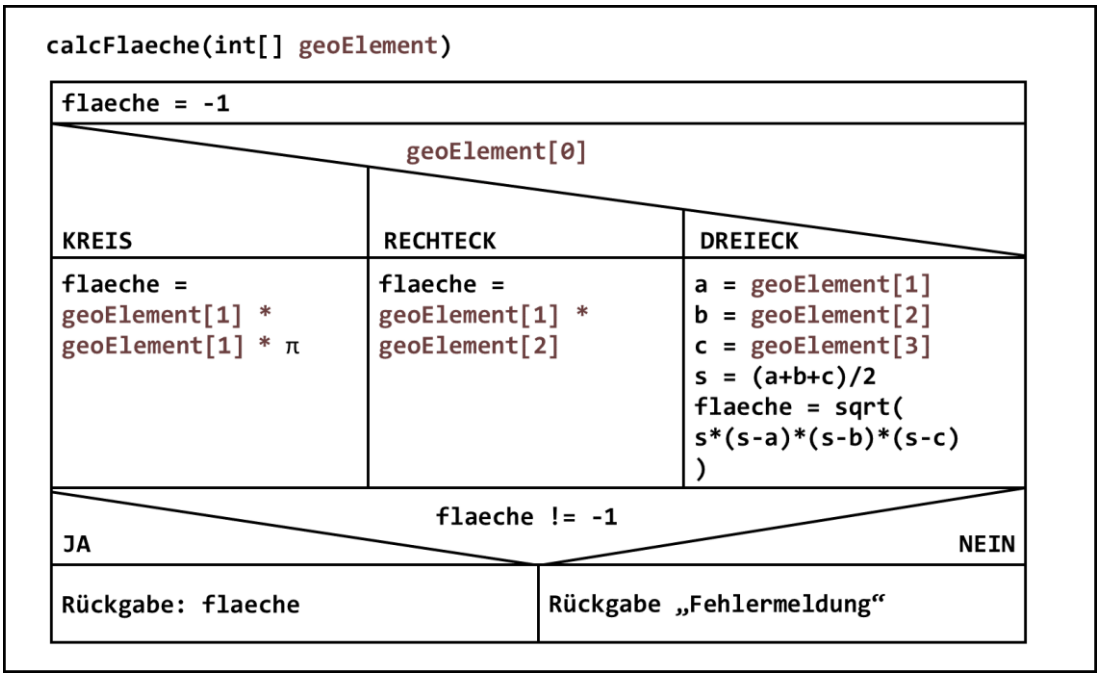
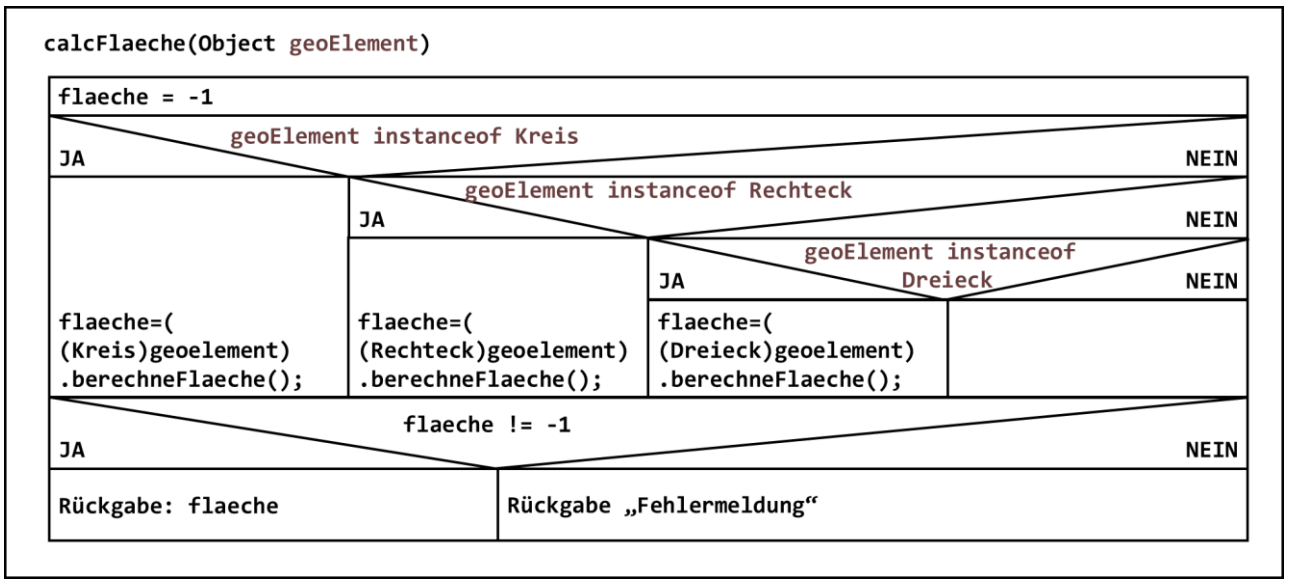
5 Funktionalität gehört zum Objekt

Gehen wir davon aus, wir wollen unser Geometrieprogramm ohne Objektorientierung umsetzen. Um dies zu realisieren, könnten wir bspw. die Informationen über unsere Geometrieobjekte in Arrays hinterlegen. An der 0. Position würde eine Integer Konstante stehen, welche Aufschluss darüber gibt, ob es sich um einen Kreis, Dreieck oder Rechteck handelt. In den folgenden Arraypositionen (1 bis n) würden für den Kreis der Radius, beim Rechteck die Länge/Breite usw. stehen. Um nun die Fläche berechnen zu können, müssten wir einen Algorithmus wie diesen programmieren:

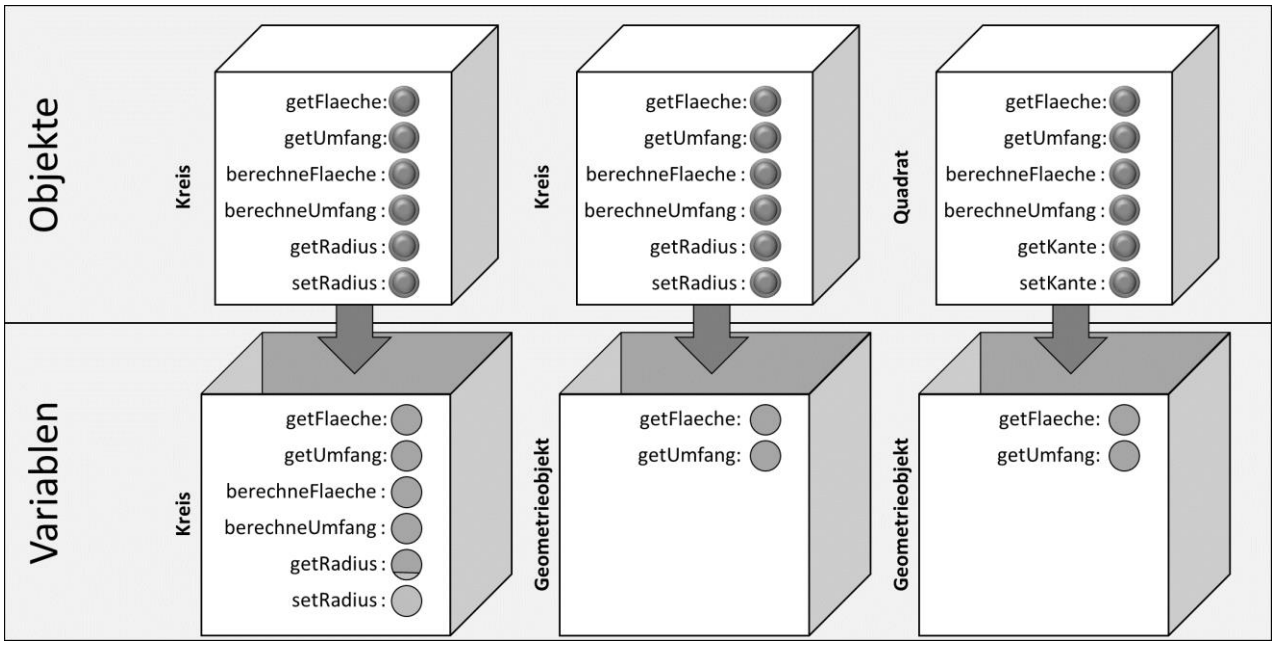


Für die Umfangberechnung hätten wir einen ähnlichen Ansatz. Wenn wir das Ganze nun objektorientiert handhaben würden, hätten wir im ersten Schritt schon mal einen Vorteil – wir würden die Berechnung der Fläche nicht in einem zentralen Unterprogramm hinterlegen, sondern in den Objekten. Der Algorithmus würde sich wie folgt ändern:



Da wir die einzelnen Flächenberechnungen in den Klassen realisiert haben, müssen wir die Formeln hierzu nicht mehr in dem Unterprogramm „`calcFlaeche()`“ halten. Wir haben aber die Funktionalität „`berechneFlaeche()`“ jeweils in den einzelnen Objekten individuell hinterlegt, also müssen wir immer prüfen, um welches Objekt es sich handelt (mit `instanceof`) und dann den korrekten Typecast durchführen. Somit haben wir immer noch eine problematische Abhängigkeit in unserem Code. Wenn wir ein weiteres Geometrieelement hinzufügen wollen (bspw. ein Quadrat), so müssten wir diese zentrale Funktion „`calcFlaeche()`“ ergänzen. Dies hätte wiederum einen Test zur Folge, wodurch wir im Prinzip die gesamte Software „als Ganzes“ im Blick haben müssen. Natürlich könnten wir die Fläche als Variable in den Objekten bereits belegen, aber dann könnten wir unseren Code nicht mehr mit der prozeduralen Lösung vergleichen. Das wäre also „unfair“ 😊. Man kann es so zwar umsetzen, allerdings haben wir dann eine redundante Datenhaltung, was mitunter auch problematisch sein kann.

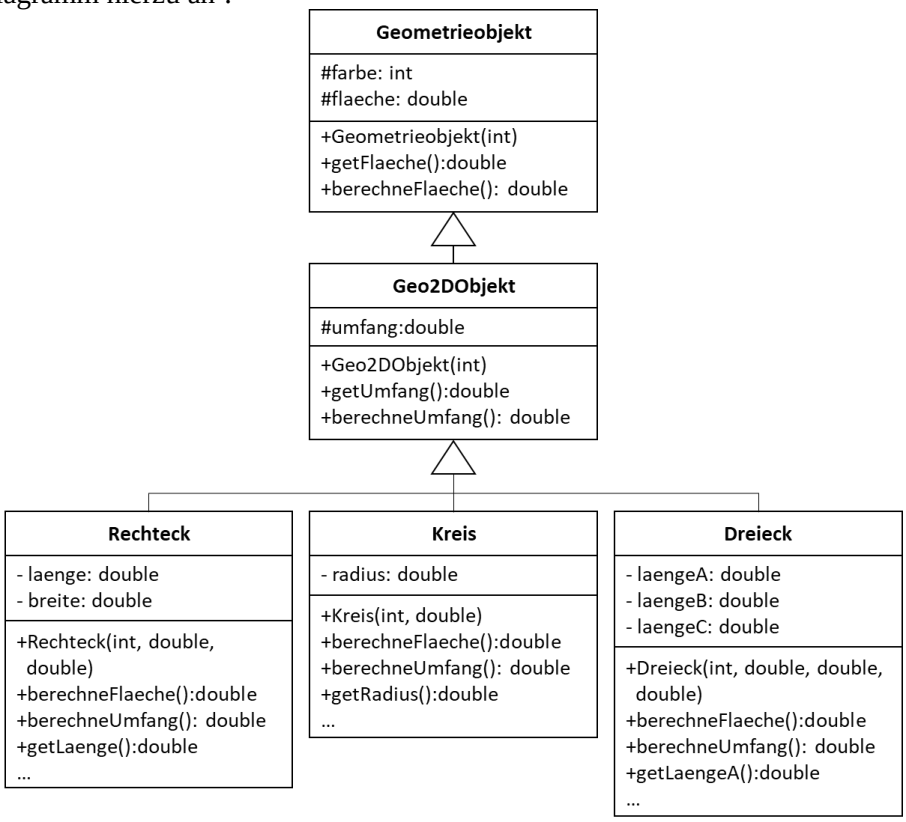
Um nun den nächsten „ultimativen“ Vereinfachungsschritt zu verstehen, müssen wir einen kleinen Exkurs zu Typecasts bei Objekten machen. Dazu hilft uns die Vorstellung, dass Objekte tatsächliche Körper sind und Variablen „lediglich“ Behälter, in denen wir die Körper hineinsetzen können. Die Objekte setzen die eigentlichen Funktionen um (vergleichbar mit „Bedienknöpfen“, welche eine Funktionalität haben) und die Variablen geben lediglich diese Funktionalitäten frei. Eine Klasse kann somit zur Definition einer Variablen oder zur Erzeugung eines Objektes dienen:



In der Grafik sehen wir also ganz Links ein Objekt vom Typ „Kreis“, welches in eine Variable vom Typ „Kreis“ gelegt wird. Da beide von der gleichen Klasse sind, gibt die Variable auch die gleichen Funktionalitäten frei, da diese ja im Objekt tatsächlich implementiert sind. Wir können somit auch das Objekt „Kreis“ in eine Variable legen, welche vom Typ „Geometrieobjekt“ ist – wie in der Mitte der Grafik. Da das Kreisobjekt alle Methoden der (Groß-) Eltern erbt, muss zwangsläufig überall dort, wo die Variable eine Methode „freigibt“ die Methode im Objekt auch vorhanden sein.

Das gibt uns nun auch die Möglichkeit, in die Variable vom Typ „Geometrieobjekt“ Objekte von allen „Nachfahren“ des Geometrieobjektes hineinzulegen – bspw. auch Quadrat, solange eben das Quadrat in der Erbfolge des Geometrieobjektes liegt. Nun liegt der Gedanke nahe, dass wir in der Variablen „Geometrieobjekt“ auch die Freigaben der Funktionen „`berechneFlaeche()`“ und „`berechneUmfang()`“ erzwingen. Dies können wir dadurch erreichen, indem wir diese Methoden bereits in der Klasse „`Geometrieobjekt`“ hinterlegen. Zeichnen Sie also die beiden Methoden in die beiden Variablen vom Typ `Geometrieobjekt` ein. Wir können die Methoden aber nicht mit sinnvollen Inhalten versehen – also lassen wir sie (erstmal) leer. Wir haben die Methode „`berechneFlaeche()`“ aus Sicht des Objektes Kreis also zweimal implementiert – einmal auf Ebene von Geometrieobjekt und einmal auf Ebene des Kreises. Dort wurde die Methode:

(Überschreiben von Methoden ist nicht zu verwechseln mit „Überladen“. Dies ist dann der Fall, wenn wir gleichen Methodennamen, jedoch unterschiedliche Parameterstrukturen haben.) Sehen wir uns das Resultierende Klassendiagramm hierzu an[1]:



(In den unteren Klassen wurden nur ein Teil der Methoden angegeben.) Nun können wir den ursprünglichen Algorithmus wie folgt realisieren:

```
calcFlaeche(Geometrieobjekt geoElement)
    Rückgabe: geoElement.berechneFlaeche()
```

Die endgültige Auslagerung der Berechnung und die Überschreibung der Berechnungsmethode hat nun zwei entscheidende Vorteile:

1. Wir haben keinerlei Änderungsnotwendigkeiten mehr im Algorithmus, wenn wir eine neue Klasse einführen.
2. Es gibt nicht mehr das Problem, dass wir ein Objekt haben, dessen Berechnungsformel uns nicht bekannt ist – weshalb das Errorhandling wegfällt.

Die Methode „`berechneFlaeche()`“ ist aus Sicht der Klasse Geometrieobjekt also „polymorph“, da eine Variable dieses Typs unterschiedliche Algorithmen bei einem entsprechenden Methodenaufruf an den Tag legt. Erst zur Laufzeit wird festgelegt, welche der Algorithmen auszuführen ist (basierend auf dem Objekt, welches in der Variable „steckt“). Dies nennt man auch

Für unseren Algorithmus heißt das nun, dass wir das Unterprogramm „`calcFlaeche()`“ eigentlich nicht benötigen.

[1] Die Zugriffsmodifikatoren (+, -, # und ~) werden später behandelt