

```
Rational r(6, 7);
const Rational CR(8, 9);
r.ausgabe();           // ruft Methode 1
CR.ausgabe();         // ruft Methode 2
```

5.6 Faustregeln zur Konstruktion von Schnittstellen

Dieser Abschnitt gibt einige Empfehlungen zur Konstruktion der Prototypen von Methoden. Zur Erinnerung: Ein Prototyp ist die Deklaration einer Schnittstelle, über die von außen auf ein Objekt zugegriffen werden kann. Ein Prototyp besteht aus Rückgabebetyp, Name und Parameterliste, wie schon aus Abschnitt 4.1.1 bekannt ist. Dazu kommt manchmal noch ein `const`-Qualifizierer, wie im vorhergehenden Abschnitt beschrieben. Unter anderem beziehen sich die Empfehlungen auf diese vier Bestandteile. Als Beispiel in einigen Fällen dient die Klasse `Geld`, deren Objekte aus einem Geldbetrag und der Währung als Attribute bestehen. Die Klasse soll in einer Anwendung Folgendes leisten:

```
// main()-Programm
Geld kaufpreis(100.00, "Euro");
cout << kaufpreis.betrag();           // 100
cout << kaufpreis.waehrung();        // Euro
// Ausgabe für Scheckvordrucke:
cout << kaufpreis.toString();        // eins-null-null
kaufpreis.neuerbetrag(90.0);         // Betrag ändern
```

Aus der Anwendung ergeben sich Prototypen, die im `public`-Bereich der Klasse aufgeführt sind. Die Namen der privaten Attribute sind beliebig gewählt.

```
class Geld {
public:
    Geld(double einBetrag, const string& eineWaehrung);
    double betrag() const;
    void neuerBetrag(double);
    const string& waehrung() const;
    string toString() const;
private:
    double derBetrag;
    string dieWaehrung;
};
```

Die Methode `toString()` berechnet aus dem Betrag einen String entsprechend der Ziffernfolge des ganzzahligen Anteils des Betrags und gibt ihn zurück. Hier nun die sechs Faustregeln:

1. Die geplante Anwendung bestimmt die Methoden. Beim Entwurf einer Klasse ist es empfehlenswert, sich die Anwendung vorher zu überlegen und nur solche Prototypen vorzusehen, die dazu beitragen. Es ist nicht sinnvoll, Methodendeklarationen aufzuschreiben, die nur *vielleicht* brauchbar sind und deren Anwendung ungewiss ist. Der Grund dafür ist einfach: Alle Methoden müssen irgendwann auch definiert, dokumentiert und getestet werden. Diese Arbeit kann man sich für alle Methoden sparen, die nicht zum Einsatz kommen.
2. Der Name einer Methode soll beschreiben, was die Methode tut. Zum Beispiel ist der Name `betrag` aussagekräftiger als etwa `zahl`.
3. Eine Methode, die den Zustand eines Objekts nicht ändert, erhält den `const`-Qualifizierer (Methoden `betrag()`, `wahrung()` und `toString()`, nicht aber `neuerBetrag()`). Siehe auch die ausführliche Diskussion in Abschnitt 5.5.
4. Der Rückgabebetyp ist natürlich `void`, wenn die Methode zwar etwas tut, aber nichts zurückgibt. Andernfalls bestimmt sich der Rückgabebetyp aus der Antwort auf die Frage: Ist der zurückgegebene Wert ein Attribut der Klasse?
 - **Ja:** Nächste Frage: Ist der zurückgegebene Wert von einem Grunddatentyp? (d.h. `int`, `double`, `bool` usw. im Gegensatz zu einem Klassentyp)
 - *Ja:* Rückgabe per Wert. Beispiel: `betrag()`
 - *Nein:* Rückgabe per `const&`. Beispiel: `wahrung()`
 - **Nein:** Rückgabe per Wert. Beispiel: `toString()`
5. Die Art der Übergabe eines Objekts in der Parameterliste kann ebenfalls durch die Beantwortung einiger Fragen bestimmt werden. Soll das Objekt beim Aufruf der Methode verändert werden?
 - **Ja:** Übergabe per Referenz (siehe Seite 122)
 - **Nein:** Nächste Frage: Gehört der übergebene Wert zu einem Grunddatentyp?
 - *Ja:* Übergabe per Wert. Beispiel: `neuerBetrag(double)`
 - *Nein:* Übergabe per `const&`. Die Übergabe des Strings im Konstruktor ist ein Beispiel.
6. Wenn Operationen verkettet werden sollen, ist das Objekt selbst per Referenz zurückzugeben. In C++ sind Anweisungen wie `a = b = c;` erlaubt und üblich, ebenso wie die schon bekannte Verkettung von Ausgaben mehrerer Variablen, etwa `cout << a << b << c << endl;`. Diese Verkettung von Operatoren wird ausführlich in Kapitel 9 diskutiert. Es gibt aber auch ein Äquivalent für Funktionen. Nehmen wir zum Beispiel die Anwendung der Methode

```
// Methode der Klasse Rational
void add(const Rational& r);
```

von Seite 180 auf rationale Zahlen a , b und c . Die Operation $a.add(b)$; ist damit möglich, nicht aber Verkettungen wie zum Beispiel

```
a.add(b).add(c); // a entspricht: a += b; a += c;
a.add(b.add(c)); // b entspricht: a += b += c;
```

Die Verkettungen könnten natürlich durch jeweils zwei einzelne Anweisungen ersetzt werden, aber hier geht es darum, zu zeigen, dass und wie Verkettungen mit der Rückgabe des Objekts selbst möglich gemacht werden. Betrachten wir beide Fälle:

- (a) Diese Anweisung wird von links abgearbeitet. $a.add(b)$ muss dann ein Objekt zurückliefern, auf das dann $add(c)$ angewendet wird. Auf einen Rückgabotyp `void` lässt sich keine Funktion anwenden, und der Compiler würde sich bitter beschweren. Dieses zurückgegebene Objekt ist sinnvollerweise nichts anderes als das durch die Addition veränderte Objekt a , hier \underline{a} genannt. Aus $a.add(b)$ ergibt sich also \underline{a} , für das $add(c)$ aufgerufen wird. In Einzelschritte zerlegt:

```
a.add(b).add(c);
    a.add(c);
```

Damit ergibt sich eine veränderte Implementation (und Deklaration) für `add()`:

```
// veränderter Rückgabotyp:
Rational& Rational::add(const Rational& r) {
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerzen();
    // Rückgabe des Objekts, für das die Methode aufgerufen wird:
    return *this;
}
```

Was bedeutet `*this`? Es ist einfach eine andere Bezeichnung für das Objekt selbst. Es muss in einer Methode so einen universellen Namen geben, weil der beliebige Name, der irgendwann in einem Anwendungsprogramm für ein Objekt vergeben wird, in der Methode unbekannt ist. Das Schlüsselwort `this` wird genauer in Abschnitt 6.10 erklärt.

- (b) Ein Aufruf der Form $a.add(x)$ verlangt, dass das Argument x vom Typ `Rational` ist. Da x identisch mit $b.add(c)$ ist und gemeint ist, dass erst c auf b addiert wird, welches dann auf a addiert wird, folgt daraus, dass der Aufruf $b.add(c)$ das veränderte b zurückgeben muss. Die unter a) angegebene Implementierung löst auch dieses Problem.