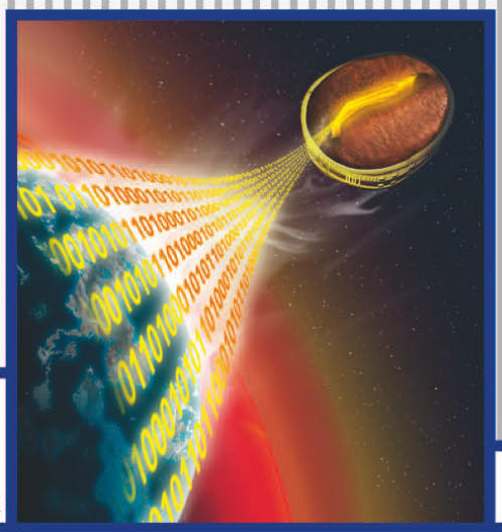
WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE



it informatik

Reinhard Schiedermeier

Programmieren mit Java

2., aktualisierte Auflage

PEARSON Studium

Man spricht Klassen, die sich hier wie die primitiven Typen verhalten, **Wertesemantik** (*value semantics*) zu. Der Begriff drückt aus, dass Operationen nur die Werte der beteiligten Objekte auslesen, sie aber ansonsten unberührt lassen. Unveränderliche Klassen haben Wertesemantik.

Referenzsemantik veränderlicher Klassen

Demgegenüber zeigen andere Klassen **Referenzsemantik** (*reference semantics*): Operationen können das Zielobjekt oder die als Operanden beteiligten Objekte verändern.

Einem beliebigen Methodenaufruf ist nicht anzusehen, ob er beteiligte Objekte manipuliert beziehungsweise welche betroffen sind. Sucht man etwa nach der Ursache für ein fehlerhaftes Objekt, dann kommt jeder Methodenaufruf in Betracht, in den das Objekt verwickelt ist. Zum Beispiel könnte der Aufruf

```
a.mult(b)
```

a oder **b** oder keines oder beide verändern.

Bei unveränderlichen Klassen vereinfacht sich das Bild drastisch: Wenn ein Objekt einmal korrekt erzeugt ist, können es nachfolgende Methodenaufrufe wegen der Wertesemantik nicht mehr stören. Die Fehlersuche reduziert sich hier allein auf die Konstruktoraufrufe.

Wertesemantik ist vorteilhaft

Aus diesem Grund sind unveränderliche Klassen und die daraus resultierende Wertesemantik vorteilhaft. Eine Klasse sollte nach Möglichkeit als unveränderlich definiert werden.[27]

Unveränderlichkeit auf logischer Ebene

Logische Unveränderlichkeit

Trotz Wertesemantik ist es manchmal sinnvoll, Objektvariablen nicht mit dem Modifier **final** zu definieren und damit technisch Veränderungen am Objekt zuzulassen. Im Allgemeinen wird von einer unveränderlichen Klasse nur gefordert, dass ihre Objekte *logisch* unveränderlich sind. Das auf die Bits und Bytes jedes Objektes zu übertragen, ist unter Umständen zu rigoros.

Zum Beispiel sind die beiden Brüche $\frac{3}{4}$ und $\frac{6}{8}$ arithmetisch gleich. Das bedeutet, dass man ein **Rational**-Objekt mit dem Inhalt $\frac{6}{8}$ kürzen kann, ohne dass sich dabei sein mathematischer Wert ändert. Die Objektvariablen erhalten dabei aber neue Werte und können folglich nicht **final** definiert sein.

Die Forderung nach **final**-Objektvariablen für unveränderliche Klassen kann also abgeschwächt werden. Im nächsten Abschnitt wird mit dem Modifier **private** ein weiteres Sprachmittel eingeführt, mit dessen Hilfe Objekte trotz veränderlicher Objektvariablen gegen unerwünschte Zugriffe abgeriegelt werden können.

27 Es gibt derzeit kein Sprachmittel in Java, mit dem Unveränderlichkeit ausgewiesen werden kann, sodass sie der Compiler überprüft und sicherstellt. Entsprechende Vorschläge liegen aber vor und werden für künftige Versionen von Java in Erwägung gezogen.

4.11 Datenkapselung

Reduktion der Abhängigkeiten zwischen Modulen

Wichtiges Hilfsmittel bei der Konstruktion von Programmen ist die Modularisierung. Ein Programm wird dabei in Teile zerlegt, die sich möglichst einzeln und unabhängig behandeln lassen. Ein Programmteil wird als „Modul“ bezeichnet.

Je weniger Abhängigkeiten zwischen verschiedenen Modulen existieren, desto leichter lassen sie sich einzeln entwickeln, testen, austauschen, korrigieren, weiterentwickeln und so fort.

Datenkapselung als Hilfsmittel

Ein wichtiges Hilfsmittel zur Reduktion der Abhängigkeiten zwischen Modulen ist die **Datenkapselung** (*data encapsulation* oder *data hiding*).

Daten und Operationen

Verbergen von Daten

Mit der Datenkapselung werden Daten hinter einem Satz von Operationen verborgen. Jeder Zugriff auf die Daten ist nur noch über die Operationen möglich, die Daten selbst sind nach außen nicht sichtbar.

Bei objektorientierten Sprachen bieten sich Klassen als Einheiten für Datenkapselung an. Die zu verbergenden Daten entsprechen den Objektvariablen, die Operationen den Methoden.

Kein Zugriff auf Objektvariablen, nur auf Methoden

Für den Benutzer einer Klasse bedeutet das, dass die Objektvariablen nicht erreichbar sind. Ihm werden nur Methoden zur Verfügung gestellt, die „in seinem Auftrag“ mit den gekapselten Objektvariablen arbeiten. Über den tatsächlichen Innenaufbau von Objekten weiß er nichts mehr und hat insbesondere auch keine Möglichkeit, sich davon abhängig zu machen.

Zugriffsschutz

private verhindert Zugriff auf Objektvariablen

In Java kann jede Objektvariable mit dem Zugriffsschutz-Modifier **private** versehen werden. Eine **private**-Objektvariable ist nur innerhalb der Klassendefinition sichtbar, aber nicht außerhalb davon.

Die folgende Definition von **Rational** verbirgt die beiden Objektvariablen:

```
class Rational {
   private int numer;
   private int denom;
   ...
}
```

Ein Zugriff auf die Objektvariablen ist von außen nicht mehr möglich:

```
class Application {
   public static void main(String[ ] args) {
      Rational r = new Rational();
      r.numer = 1;                  // unzulässig
   }
}
```

Freier Zugriff innerhalb der eigenen Klasse

Die Methoden der Klasse selbst, wie zum Beispiel **output** und **reduce**, können dagegen weiterhin ohne Einschränkung mit den Objektvariablen arbeiten.

Der Schutz mit dem Modifier **private** gilt für den Compiler beim Übersetzen von Java-Programmen. Er existiert nicht für die JVM, das heißt beim Ablauf des Programms. Zum Beispiel hat die Methode **mult**, ob destruktiv (Seite 130) oder mit Wertesemantik (Seite 142), weiterhin freien Zugriff auf die Objektvariablen des *anderen* **Rational**-Objektes, das im Parameter **p** übergeben wird. **private** und alle weiteren Zugriffsschutz-Modifier beziehen sich auf ganze Klassen, nicht auf einzelne Objekte.

private-Methoden nicht von außen aufrufbar

Auch Methoden können dem Modifier **private** versehen werden. Für Methoden, die dem Anwender einer Klasse zur Verfügung gestellt werden sollen, ist das natürlich sinnlos. Allerdings kann es durchaus Hilfsmethoden geben, die nur zur Nutzung ausschließlich innerhalb der Klasse vorgesehen sind. Diese wird man mit **private** gegen Zugriff von außen schützen.

Die Klasse **Rational** könnte zum Beispiel eine private Hilfsmethode definieren, die den größten gemeinsamen Teiler von zwei Zahlen berechnet. Dem Anwender der Klasse muss diese nicht unbedingt zur Verfügung gestellt werden, andere Methoden der Klasse können sie aber vielleicht gut gebrauchen.

Getter

Getter ersetzen lesenden Zugriff auf Objektvariablen

Eine mit dem Modifier **private** geschützte Objektvariable ist von außen nicht mehr erreichbar. Wenn man den Wert dennoch zum Lesen zur Verfügung stellen möchte, bietet man eine Auskunftsmethode an, die als „Getter-Methode“ oder kurz als **Getter** bezeichnet wird, manchmal auch als „Inspektor-“ oder „Accessor-Methode“.

Definitionsschema für Getter

Ein Getter lässt sich fast mechanisch schreiben. Wenn eine Objektvariable definiert ist als

```
private type name;
```

lautet der entsprechende Getter:

```
type getName()
{
    return name;
}
```

Man könnte einem Getter zwar jeden beliebigen Namen geben, aber eine Bezeichnung nach dem Schema **get***Name* hat sich eingebürgert und wird inzwischen von vielen Werkzeugen unterstellt. Dabei ist *Name* der Name der Objektvariablen, wobei der erste Buchstabe großgeschrieben ist.

In der Definition auf Seite 138 sind beispielsweise die Getter **getNum** und **getDenom** für die Objektvariablen **numer** und **denom** der Klasse **Rational** definiert.

Setter

Setter zum Verändern von Objektvariablen

Bei unveränderlichen Klassen können die Objektvariablen nicht modifiziert werden. Bei anderen Klassen werden Änderungen zugelassen. Nachdem der direkte Zugriff auf die Objektvariablen mit dem Modifier **private** abgeriegelt ist, stellt man Änderungsmethoden zur Verfügung, die als „Setter-Methoden“ oder **Setter** bezeichnet werden (manchmal auch „Modifier-Methoden“), entsprechend zu Gettern.

Definitionsschema für Setter

Auch ein Setter kann nach einem festen Schema geschrieben werden. Zu einer Objektvariable mit der Definition

```
private type name;
```

definiert man einen Setter als:[28]

```
void setName(type name)
{
   this.name = name;
}
```

Die Benennung **set***Name* folgt der gleichen Logik wie die Bezeichnung der Getter im vorhergehenden Abschnitt.

Das folgende Beispiel zeigt die zwei Setter **setNum** und **setDenom** einer veränderlichen Fassung der Klasse **Rational**:

```
class Rational {
   private int numer;
   private int denom;

   void setNum(int n) {
      numer = n;
   }

   void setDenom(int d) {
      denom = d;
   }
   ...
}
```

Kontrolle der Zugriffe

Man könnte den Eindruck gewinnen, dass eine Klasse mit privaten Objektvariablen, Settern und Gettern zwar komplizierter ist als eine Klasse mit ungeschützten, öffentlichen Objektvariablen, aber weiter keine Vorteile bietet. Selbst unveränderliche Klassen könnte man mit öffentlich erreichbaren Objektvariablen definieren, indem man mit dem Modifier **final** Änderungen blockiert.

28 Hier ist this (siehe Seite 125) nötig, um die Objektvariable mit dem Namen *name* zu erreichen, die vom Parameter mit dem gleichen Namen verdeckt wird.

Es gibt trotzdem gute Gründe, Zugriffe ausschließlich über Methoden zu gestatten und nicht direkt auf Objektvariablen zuzugreifen:

Kontrolle der Werte

- Kaum jemals ist der gesamte Wertebereich des Typs einer Objektvariablen als Inhalt zulässig. Im Beispiel der Klasse **Rational** darf der Nenner keinesfalls den Wert null haben. Weiter könnte man einige **Rational**-Methoden vereinfachen, wenn nur der Zähler negativ sein darf, der Nenner aber immer positiv bleibt. Würde man freien Zugriff auf die Objektvariablen erlauben, dann gäbe es keine Möglichkeit, die Zuweisung unerwünschter Werte zu verhindern. Setter können dagegen ihre Parameter prüfen und gegebenenfalls Werte ändern oder abweisen.

Hilfsmittel zur Fehlersuche

- Bei der Fehlersuche muss man oft der Ursache für Objekte mit inkonsistentem Zustand auf die Spur kommen. Der direkte Zugriff auf eine Objektvariable lässt sich nicht überwachen und kann jederzeit im Programm erfolgen. In Gettern und Settern kann man dagegen Ausgabeanweisungen einfügen und erhält dann ein komplettes Protokoll aller Zugriffe samt der dabei übermittelten Werte.

Setter/Getter für scheinbare Objektvariablen

- Eigenschaften können durch Getter und Setter vorgespiegelt werden, obwohl es in der Klassendefinition keine entsprechenden Objektvariablen gibt. In einer anderen Version der **Rational**-Klasse könnte der Wert eines Bruchs intern in einer **double**-Objektvariablen[29] gespeichert werden, zum Beispiel 0.75 für $\frac{3}{4}$. Die Getter **getNum** und **getDenom** berechnen dann bei jedem Aufruf Zähler und Nenner neu aus dem Dezimalbruch. Von außen ist nicht erkennbar, ob die Getter ihre Auskunft aus Objektvariablen lesen oder berechnen.

Interne Maßnahmen verbergen

- Mit Gettern und Settern können interne Optimierungen verborgen werden. Zum Beispiel könnte man in der Klasse **Rational** Zähler und Nenner so lange ungekürzt belassen, bis der Anwender einen Getter aufruft. Zusätzlich könnte eine private **boolean**-Objektvariable aufzeichnen, ob der Bruch seit dem letzten Kürzen verändert wurde und ob deshalb erneutes Kürzen überhaupt notwendig ist. Aus der Sicht des Anwenders erscheinen Brüche immer gekürzt. Er kann nicht erkennen, dass sie immer erst im letzten Moment und dann auch nur bei Bedarf gekürzt werden.

Schnittstellen

Trennung zwischen Schnittstelle und Implementierung

Um mit einer Klasse zu arbeiten, braucht ein Anwender die Rümpfe von Methoden nicht zu kennen. Es reicht zum Beispiel aus, wenn er von einer Methode **reduce** weiß, dass sie einen Bruch kürzt. *Wie* die Methode das im Detail bewerkstelligt, ist unerheblich.

Als „Schnittstelle“ einer Klasse wird die Gesamtheit aller Informationen bezeichnet, die ein Anwender braucht, um die Klasse zu verwenden. Alles andere zählt zur „Implementierung“ der Klasse.

Aus der Sicht einer Klassendefinition zählen nur die öffentlichen Methodenköpfe und Objektvariablen zur Schnittstelle, aber weder Methodenrümpfe noch private Methoden und Objektvariablen.

29 Die Idee ist in diesem konkreten Beispiel nicht unproblematisch wegen der begrenzten Genauigkeit von Gleitkommawerten.

Klassennutzung als Geschäftsbeziehung

Die Beziehung zwischen der Definition einer Klasse und ihrer Verwendung ähnelt in vieler Hinsicht einer Geschäftsbeziehung zwischen einem Anbieter und seinen Kunden.

Die Klasse entspricht dem Anbieter, ihre Anwender den Kunden. Anbieter und Kunden vereinbaren Leistungen in einem Vertrag. Dieser Vertrag ist die Schnittstelle der Klasse.

Die Implementierung der Klasse entspricht internen Maßnahmen und Betriebsmitteln des Anbieters, die er benutzt, um die vereinbarten Leistungen zu erbringen. Für den Kunden sind sie unerheblich, er ist nur am Ergebnis interessiert.

Wenn ein Anbieter zugesicherte Leistungen ändert oder sogar streicht, sind davon alle Kunden betroffen. Dementsprechend wird man Modifikationen an der Schnittstelle einer Klasse möglichst vermeiden.

Für die internen Maßnahmen, also für die Implementierung der Klasse, gilt das nicht. Diese können ohne Rücksprache und ohne Auswirkungen auf die Kunden manipuliert werden.

Wie in einer Geschäftsbeziehung wird ein Anbieter in einem Vertrag nicht mehr als die notwendigen Verbindlichkeiten festschreiben. In einer Klassendefinition wird man deshalb alle Objektvariablen und Methoden mit dem Modifier **private** definieren, die über die vereinbarte Schnittstelle hinausgehen.

4.12 Klassenvariablen

Statische Elemente einer Klasse als Ganzes zugeordnet

Die bisher betrachteten Objektvariablen und Methoden beziehen sich immer auf ein bestimmtes Objekt, das jeweilige Zielobjekt. Demgegenüber sind Klassenvariablen und statische Methoden einer Klasse als Ganzes zugeordnet und nicht einzelnen Objekten. Das bedeutet, dass es zum Beispiel nur *ein einziges* Exemplar einer Klassenvariablen gibt, das sich alle Objekte der Klasse teilen.[30]

Zuerst werden Klassenvariablen diskutiert, auf statische Methoden wird im nächsten Abschnitt eingegangen.

Definition

Definition einer Klassenvariablen

Die Definition einer Klassenvariablen folgt dem gleichen Schema wie die Definition einer Objektvariablen. Zusätzlich wird der Modifier **static** vorangestellt, wie im folgenden Beispiel:

30 Klassenvariablen und statische Methoden entsprechen globalen Variablen und Funktionen älterer Programmiersprachen. Unabhängig von der Syntax können damit in Java Programme geschrieben werden, die alle Vorteile objektorientierter Sprachen verlieren und strukturell auf der Ebene beispielsweise von C-Programmen stehen.

```
class Rational {
    static int count;
    ...
}
```

Diese unscheinbare Änderung hat weitreichende Folgen für die Variable.

Zugriff

Zugriff mit Klassennamen, ohne Zielobjekt

Eine Klassenvariable ist unabhängig von Objekten der betreffenden Klasse. Es spielt keine Rolle, ob oder wie viele Objekte existieren. Nachdem es vielleicht überhaupt kein Objekt der Klasse gibt, kann ohne Objekt auf eine Klassenvariable zugegriffen werden. Stattdessen wird eine Klassenvariable über den Klassennamen angesprochen,[31] wie beispielsweise:

```
Rational.count = 0;
```

Klassenvariablen sind schon weiter vorne vorgekommen (beispielsweise Seiten 56 und 59). Die beiden Ausdrücke

```
Math.PI
Integer.MAX_VALUE
```

sprechen die Klassenvariable **PI** der Klasse **Math** beziehungsweise die Klassenvariable **MAX_VALUE** der Klasse **Integer** an.

Initialisierung

Initialisierung

Ebenso wie normale Objektvariablen können Klassenvariablen bei der Definition initialisiert werden:

```
class Rational {
    static int count = 0;
    ...
}
```

Lebensdauer gesamte Programmlaufzeit

Ohne explizite Initialisierung werden Klassenvariablen mit dem typabhängigen Defaultwert vorbesetzt (siehe Seite 134). Die Klassenvariable **count** im vorhergehenden Beispiel würde auch ohne Initialisierung mit dem Wert **0** starten. Klassenvariablen werden geschaffen, wenn eine Klasse zum ersten Mal in irgendeiner Form angesprochen wird, und existieren dann bis zum Ende des Programms.[32]

---

31 Eine Klassenvariable kann alternativ auch syntaktisch wie eine normale Objektvariable über ein Objekt der Klasse als Zielobjekt angesprochen werden. Welches Objekt dazu benutzt wird, spielt keine Rolle, weil immer dieselbe Klassenvariable erreicht wird.

32 Klassen werden von der JVM dynamisch geladen. Das muss nicht notwendigerweise beim Programmstart sein, sondern kann auch zu einem späteren Zeitpunkt geschehen, wenn zum ersten Mal Bezug auf die Klasse genommen wird. Beim Laden der Klasse werden die Klassenvariablen erzeugt.

Konstanten

Anwendung Klassenvariablen: öffentliche Konstanten

Ein wichtiger Zweck von Klassenvariablen ist die Definition von Variablen, die im gesamten Programm, das heißt für alle Klassen und Methoden, gleichermaßen gelten. **Math.PI** und **Integer.MAX_VALUE** sind Beispiele dafür.

In der Regel sollen sich derartige allgemeingültige Werte nicht ändern. Die entsprechenden Variablen sind deshalb mit den Modifiern **static** und **final** definiert. Einen Auszug aus der Definition der Klasse **Integer** kann man sich folgendermaßen vorstellen:

```
class Integer {
   static final int MAX_VALUE = 2147483647;
   static final int MIN_VALUE = -2147483648;
   ...
}
```

Konvention zur Benennung von Konstanten

Unveränderliche, allgemein verfügbare Klassenvariablen werden auch als **öffentliche Konstanten** bezeichnet. Ihre Namen werden per Konvention ganz mit Großbuchstaben geschrieben, Wortteile werden mit Unterstrichen (_) getrennt.[33] **Math.PI** und **Integer.MAX_VALUE** sind öffentliche Konstanten.

Konstanten müssen sofort bei der Definition initialisiert werden, weil sie einerseits als **final**-Variablen nicht ohne Wert bleiben können und andererseits kein Konstruktor zuständig ist.

Zugriff aus Methoden

Umgang mit Klassenvariablen

Methoden können Klassenvariablen genauso ansprechen wie Objektvariablen. Es gibt aber nur ein einziges Exemplar jeder Klassenvariablen, das sich sämtliche Objekte der Klasse teilen.

Im folgenden Beispiel wird die Klassenvariable **count** im Konstruktor der Klasse **Rational** inkrementiert:

```
class Rational {
   static int count = 0;
   private final int numer;
   private final int denom;

   Rational() {
      numer = 0;
      denom = 1;
      count++;    // ebenso in allen anderen Konstruktoren
   }
   ...
}
```

33 Das weicht von der sonst in Java üblichen Schreibweise von Bezeichnern ab und hat historische Gründe.

Anwendung: Objektzähler

Da es nur ein Exemplar der Klassenvariablen **count** gibt, erhöht jeder Konstruktoraufruf den Wert dieser einen Klassenvariablen. Mit dem Ausdruck

```
Rational.count
```

kann in einer Anwendung der **Rational**-Klasse jederzeit abgerufen werden, wie viele **Rational**-Objekte bisher erzeugt wurden:

```
System.out.printf("%d rationals created.%n", Rational.count);
```

Anwendung: eindeutige Kennnummern

Eine andere Anwendung sind Seriennummern für Objekte, ähnlich wie Fahrgestellnummern in Autos. Die Klassenvariable **nextSerial** in der Klasse **Rational** speichert die jeweils nächste freie Seriennummer. Im Konstruktor wird diese nächste freie Seriennummer an die normale Objektvariable **serial** zugewiesen. Nachdem die Seriennummer damit vergeben und nicht mehr frei ist, wird die Klassenvariable für den nächsten Konstruktoraufruf inkrementiert:

```
class Rational {
   static int nextSerial = 0;
   private final int serial;
   private final int numer;
   private final int denom;

   Rational() {
      numer = 0;
      denom = 1;
      serial = nextSerial;
      nextSerial++;
   }

   int getSerial() {
      return serial;
   }
   ...
}
```

Jedes Objekt hat eine eigene, eindeutige Seriennummer, die in einer normalen Objektvariablen gespeichert ist und mit einem Getter abgerufen werden kann:

```
Rational r = new Rational();
...
System.out.printf("This is rational no. %d%n", r.getSerial());
```

Die Seriennummer ist mit dem Modifier **final** vor Veränderungen und mit **private** vor Zugriff geschützt. Die Konsistenz der Seriennummern ist damit sichergestellt.

4.13 Statische Methoden

Neben Klassenvariablen können auch **statische Methoden** definiert werden. Wie Klassenvariablen sind statische Methoden unabhängig von Objekten und „gehören“ der ganzen Klasse. Die Definition einer statischen Methode wird mit dem Modifier **static** markiert, ebenso wie Klassenvariablen:

Statische Methoden laufen ohne Objekt

```
class Rational {
   static int getCount() {...}
}
```

Im Übrigen gelten für die Definition statischer Methoden alle Freiheiten normaler Methoden: Der Zugriff kann mit dem Modifier **private** eingeschränkt werden, sie können überladen werden, Parameter haben und Ergebnisse liefern.

Zugriffsschutz und Überladen statischer Methoden

Der Aufruf einer statischen Methode richtet sich an die Klasse, nicht an ein Objekt:

```
int c = Rational.getCount();
```

Aufruf mit der Klasse

main-Methode

Eine statische Methode, die schon von Anfang an benutzt wurde, ist das sogenannte „Hauptprogramm“, das als Methode **main** definiert ist:

main als Beispiel einer statischen Methode

```
class SomeClass {
   public static void main(String[ ] args) {
      ...
   }
}
```

main muss statisch sein, weil beim Start des Programms noch keine Objekte existieren,[34] deren Methoden aufgerufen werden könnten. In vielen Anwendungen erzeugt **main** nur ein erstes Objekt und ruft eine Methode dieses Objektes auf. Alles Weitere entwickelt sich aus diesem Methodenaufruf.

Die Methode **main** spielt in einer Hinsicht eine Sonderrolle: Sie wird beim Start eines Java-Programms von der JVM in der Klasse gesucht, die auf der Kommandozeile angegeben ist. Das Kommando

main wird von der JVM automatisch gestartet

```
java classname
```

sucht also nach der Methode

```
class classname
{
    public static void main(String[ ] args) {...}
}
```

und ruft diese auf. Wenn sie nicht gefunden wird, scheitert der Programmstart.

34 Genau genommen arbeitet die JVM selbst mit Objekten, die bereits vor dem Aufruf von main existieren. Darauf hat eine normale Anwendung aber keinen Zugriff.

In jeder anderen Beziehung ist **main** eine gewöhnliche statische Methode. Das heißt zum Beispiel, dass man **main** überladen, im Programm von anderer Stelle selbst aufrufen und in mehreren Klassen definieren kann.

Zugriffsmethoden

Zugriffsschutz sinnvoll

Ebenso wie Objektvariablen sollten auch Klassenvariablen mit dem Modifier **private** vor direktem Zugriff geschützt werden. Der Inhalt einer Klassenvariablen lässt sich über einen „statischen Getter" abrufen:

```
class Rational {
   private static int count = 0;

   static int getCount() {
      return count;
   }
   ...
}
```

Diese statische Methode kann jederzeit benutzt werden, insbesondere auch bevor überhaupt Objekte erzeugt wurden. Vor dem ersten Konstruktoraufruf liefert sie korrekt die Anzahl **0** zurück:

```
public static void main(String[ ] args) {
   System.out.println(Rational.getCount());    // gibt "0" aus
   new Rational();
   System.out.println(Rational.getCount());    // gibt "1" aus
}
```

Statische Hilfsmethoden

Hilfsmethoden ohne Bezug zum Objekt

Manche Methoden berechnen Ergebnisse oder erbringen Leistungen, die nichts mit Objekten der Klasse zu tun haben, also auf keine Objektvariablen und Methoden der Klasse Bezug nehmen. Diese Methoden sind Kandidaten für statische Definitionen.

Beispielsweise berechnet die **Rational**-Klasse den größten gemeinsamen Teiler von Zähler und Nenner, um den Bruch zu kürzen. Die Berechnung des ggT kann von einer eigenen Methode erledigt werden, die auch ohne Brüche nützlich und sinnvoll ist. Sie kann als statische Methode definiert werden:

```
class Rational {
   static int gcd(int a, int b) {...}
}
```

Einige vordefinierte Klassen, wie zum Beispiel **Arrays** (Seite 235) und **Collections** (Seite 394), definieren überhaupt nur statische Methoden. Diese Klassen sind nicht mehr als „Behälter", welche Sammlungen verwandter statischer Methoden gruppieren. Objekte dieser Klassen sollen und können überhaupt nicht erzeugt werden.