

Inhalt

| | | |
|-------|---|------|
| 8 | Das objektorientierte Programmierparadigma | 8-2 |
| 8.1 | Strukturierung im Großen - Modularisierung..... | 8-2 |
| 8.2 | Modularisierung..... | 8-5 |
| 8.2.1 | Objekte zur Kapselung von Attributen und Methoden | 8-5 |
| 8.2.2 | Klassen zur Datenabstraktion..... | 8-6 |
| 8.3 | Vererbung, Generalisierung und Spezialisierung..... | 8-9 |
| 8.3.1 | Abgeleitete Klassen - Spezialisierung..... | 8-9 |
| 8.3.2 | Überschreiben von Methoden | 8-10 |
| 8.3.3 | Abstrakte Klassen - Generalisierung..... | 8-11 |
| 8.3.4 | Einfach- und Mehrfachvererbung | 8-13 |
| 8.4 | Polymorphismus und dynamisches Binden..... | 8-15 |
| 8.5 | Zusammenfassung - Konzepte der OOP | 8-17 |
| 8.6 | Methoden der Klasse <code>java.lang.String</code> | 8-18 |

8 Das objektorientierte Programmierparadigma

Die bisher betrachtete Art der Programmierung behandelte vorrangig die *imperativen Bestandteile* einer Sprache. Jetzt soll die *objektorientierte Programmierung* (OOP) im Vordergrund der Betrachtungen stehen.

8.1 Strukturierung im Großen - Modularisierung

Ursache

Enorme **Kostensteigerung** bei der Softwareentwicklung

| | |
|------|---------------------------|
| 1985 | 140 Milliarden Dollar |
| 1990 | 250 Milliarden Dollar |
| 2000 | ca. 800 Milliarden Dollar |

Entwurfsziel

- **Einsatz moderner Programmiermethoden**, Programmierung als *Abbild der realen Welt*.
- **Programmentwicklung nach dem Baukastenprinzip**, zur Wiederverwendung von *Software-Bausteinen (Modulen)*.
- Produktivitätssteigerung durch **Einsatz von vorgefertigten Programmierertools**, Unterstützung durch grafische Komponenten, automatische Codegenerierung, einfacher Zugang zu *Klassenbibliotheken*.

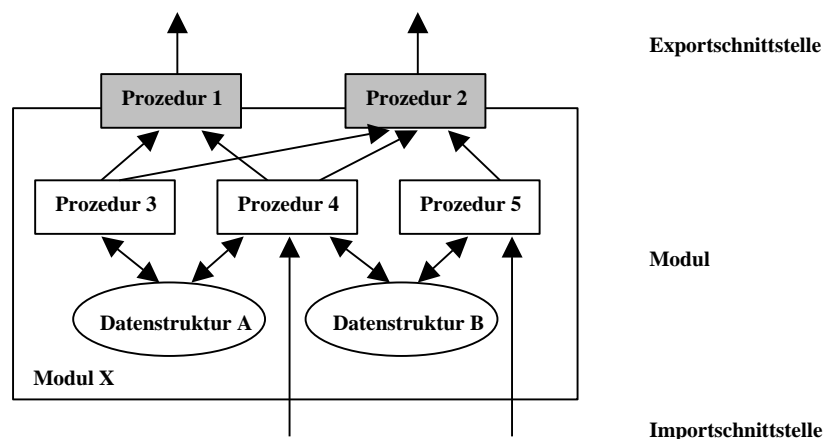
Einsatz moderner Programmiermethoden

Die *prozedurorientierte Strukturierung* hat sich als nicht mehr *ausreichend* erwiesen. Die Architektur großer Programmsysteme beruht jetzt auf **Modularisierung**. Man fasst zur Lösung eines Teilproblems mehrere *Algorithmen* und die von ihnen genutzten *Datenstrukturen* zu einer *Einheit*, einem **Modul**, zusammen.

Das Zusammenspiel mehrerer solcher *Module* wird analog der prozedurorientierten Programmierung durch ihre **Schnittstellen** geregelt. Die **Exportschnittstelle** gibt an, welche Daten ein Modul anderen Modulen zur Verfügung stellt, *welche Dienstleistung es anbietet*. Die **Importschnittstelle** bestimmt, welche Daten das Modul von anderen Modulen nutzt, *welche Dienstleistung es nachfragt*. Die Schnittstellen sind der **öffentlichen** Teil eines Moduls. Ähnlich den Prozeduren gibt es einen **nicht öffentlichen** Teil.

Schnittstelle eines Moduls

Importschnittstelle (*Nachfragen nach Dienstleistungen*) und
Exportschnittstelle (*Anbieten von Dienstleistungen*)



Programmentwicklung nach dem Baukastenprinzip

Diese Vorgehensweise entspricht dem **modularen Ansatz**. Ein Modul kann analog den Prozeduren ohne Einfluss auf die Programmierumgebung *entwickelt, getestet* und u. U. auch durch ein anderes *ausgetauscht* werden, solange die Schnittstellen selbst unverändert bleiben. Diese Art der Kapselung unterstützt die *Entwicklung* und *Wiederverwendbarkeit* von Software-Bausteinen.

Ein Modul ist eine Sammlung von Algorithmen und Datenstrukturen zur Bearbeitung einer in sich abgeschlossenen Aufgabe.

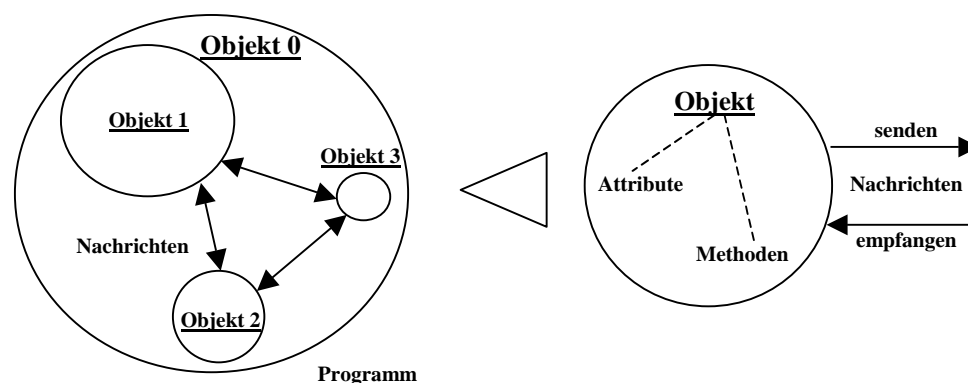
Die Integration eines Moduls in ein Programmsystem wird über seine Schnittstellen geregelt und erfordert keine Kenntnisse des Aufbaus und der konkreten Realisierung der gekapselten Datenstrukturen und Algorithmen.

Adele Goldberg

- Leiterin eines Entwicklungslabors im Xerox Palo Alto Research Center (PARC)
- „Mutter“ der objektorientierten Programmierung; Ziel ist es, „Barrieren zwischen Mensch und Maschine niederzureißen.“
- **Smalltalk** (1972), erste OOP-Sprache.
- 1987: „Oskar“ der Informatik, begehrter *Systems Software Award* der *Association for Computing Machinery (ACM)*.

Objektorientierte Programmierung

Objekte sind die Datengrundbausteine (Module), aufgebaut aus **Attributen** (Daten) und **Methoden** (Anweisungen). Objekte kommunizieren miteinander über **Nachrichten**.



Ein objektorientiertes Programm ist ein System miteinander kommunizierender Objekte, d.h. sie selbst senden, empfangen und reagieren auf Nachrichten.

Schnittstelle eines Objektes: *Nachrichten, spezielle öffentliche zugängliche Methoden.*
Importschnittstelle (Empfangen von *Nachrichten*)
Exportschnittstelle (Senden von *Nachrichten*)

OOP-Sprachen

Die Anwendung der *objektorientierten Methodik* in der industriellen Softwareentwicklung hat sich inzwischen durchgesetzt. Jeder Programmierer erzeugt seinen ihm zugeordneten *Bausatz*.

Anschließend werden diese innerhalb eines komplexen Moduls zu einem Ganzen zusammengefügt, wobei die Vorgehensweise nach dem *Baukastensystem* erfolgt.

Es existieren verschiedene OOP-Sprachen, auf imperative Sprachen aufgesetzt, die dann auch oft eine gemischte (imperative und objektorientierte) Programmierung zulassen, *Hybridsprachen*, aber auch völlig neuentwickelte, *reine* OOP-Sprachen.

| Sprache | Jahr | Eigenschaften |
|----------------|-------------|--|
| Smalltalk | 1972/80 | einfache Syntax, <i>rein objektorientiert</i> |
| Object Pascal | 1986 | Erweiterung von Pascal |
| C++ | 1986 | Weiterentwicklung von C |
| Java | 1995 | an C++ orientiert, für verteilte Systeme (Internet) geeignet |

Java, Object Pascal und **C++** sind Hybridsprache. *Einfache Datentypen* sind hier keine Objekte spezieller Klassen, die Sprachen haben sowohl *imperative* als auch *objektorientierte* Bestandteile

8.2 Modularisierung

8.2.1 Objekte zur Kapselung von Attributen und Methoden

Imperative Sprachen unterstützen den herkömmlichen prozedurorientierter Ansatz:

Ein *komplexes Problem* wird schrittweise in genügend kleine *Teilprobleme* zerlegt (*Top-Down-Analyse*). Jedes dieser Teilprogramme wird als Prozedur in einer zur Verfügung stehenden Programmiersprache implementiert und dann in einer Hauptprozedur zu einem Programm zusammengefasst.

Objektorientierte Sprachen simulieren das menschliche Denkverhalten:

Das zu modellierende *Gesamtsystem* wird zerlegt in *Teilsysteme*, bestehend aus miteinander kommunizierenden Modulen. Jedem solchen Modul wird ein *Objekt* zugeordnet. Die Objekte kommunizieren miteinander, d.h. sie selbst senden, empfangen und reagieren auf *Nachrichten*.

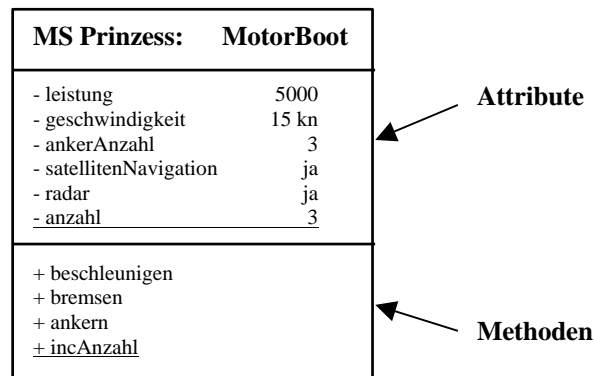
Ein **Objekt** im Sinne von OOP besteht aus:

- *Attribute:* *Daten* eines Objektes
(Strukturkomponente des Objekts).
- *Methoden:* *Operationen* zur Manipulation der Daten des Objekts
(Funktionskomponente des Objekts).

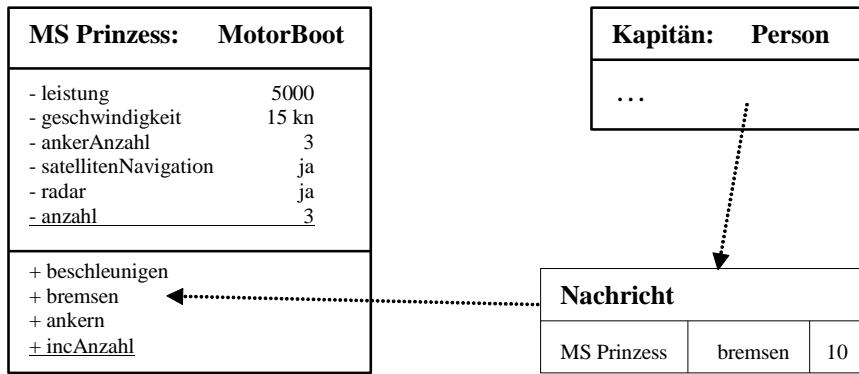
Dieses Zusammenfassen von *Attributen* und *Methoden* wird als **Kapselung** (*information hiding*) bezeichnet, eine der wichtigsten Eigenschaften der OOP. Attribute und Methoden werden als ein *Modul* behandelt. **Schnittstellen** regeln die Interaktion mit anderen Objekten. Sie bestehen aus *öffentlich zugänglichen Methoden*. Die *Attribute* eines Objekts als gekapselten Datenstrukturen sollten *nicht öffentlich* und *nur durch eigene Methoden* manipulierbar sein. Eine Veränderung der Attribute von außen *widersprüche* dieser Programmierphilosophie.

Jedes Objekt verwaltet sich selbst!

Ein Motorboot „MS Prinzess“ wird grafisch folgendermaßen umschrieben:



Zur Kommunikation mit anderen Objekten werden **Nachrichten (Botschaften)** verschickt, die bei den Objekten eine Ausführung einer der *öffentlich zugänglichen Methoden* auslösen und eine Änderungen seiner Attribute nach sich ziehen können.



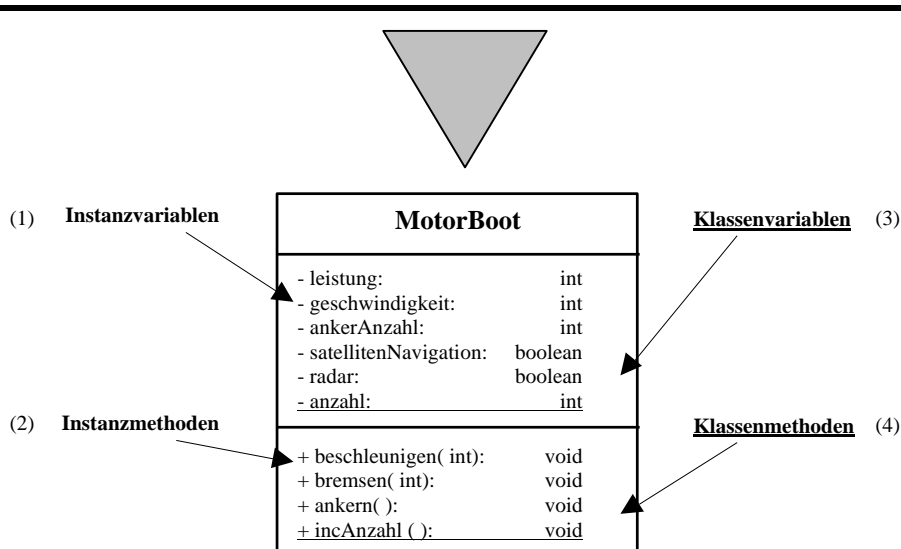
8.2.2 Klassen zur Datenabstraktion

Klassen sind *Datentypdeklarationen* im üblichen Sinn und legen Attribute und Methoden ihrer Objekte fest.

Objekte (Instanzen)

| MS Prinzess: MotorBoot | MS Rostock: MotorBoot | MS Karibik: MotorBoot |
|---------------------------|---------------------------|---------------------------|
| - leistung 5000 | - leistung 2000 | - leistung 3000 |
| - geschwindigkeit 15 kn | - geschwindigkeit 10 kn | - geschwindigkeit 10 kn |
| - ankerAnzahl 3 | - ankerAnzahl 2 | - ankerAnzahl 2 |
| - satellitenNavigation ja | - satellitenNavigation ja | - satellitenNavigation ja |
| - radar ja | - radar ja | - radar ja |
| - anzahl 3 | - anzahl 3 | - anzahl 3 |
| + beschleunigen | + beschleunigen | + beschleunigen |
| + bremsen | + bremsen | + bremsen |
| + ankern | + ankern | + ankern |
| + incAnzahl | + incAnzahl | + incAnzahl |

Klasse



- (1) **Instanzvariablen:** *objektspezifische Attribute*, die Daten eines Objekts
- (2) **Instanzmethoden:** *objektspezifische Methoden*, zur Manipulation der Objektdaten
- (3) **Klassenvariablen:** *klassenspezifische Attribute*, nicht vom Objekt aus manipulierbar
- (4) **Klassenmethoden:** *klassenspezifische Methoden*, zur Manipulation der Klassendaten

UML ... Unified Modeling Language

Klassendiagramme veranschaulichen den Aufbau einer Klasse. Neben den Klassendiagrammen gibt es noch eine Fülle weiterer Diagramme. Durch grafische Darstellungsformen werden Objektzustände, Abläufe und Beziehungen von Objekten und Klassen beschrieben.

Wir beschränken uns hier aber auf die Verwendung von Klassendiagrammen.

Das *Klassendiagramme* für eine Klasse `MotorBoot` ist folgendermaßen zu interpretieren: Unter dem Klassennamen stehen alle *Attribute*, nach dem Doppelpunkt wird ihr Typ festgelegt. Anschließend werden alle *Methoden*, einschließlich der Parameterliste und dem Ergebnistyp, aufgelistet. Das *Minus* steht für *nicht öffentlichen* und das *Plus* für *öffentlichen* Zugriff. *Klassendaten* und *Klassenmethoden* werden *unterstrichen*.

Klassendeklaration „Motorboot“**MotorBoot.java**

```
public class MotorBoot
{
// Instanzvariablen
    private int leistung;           // in PS
    private int geschwindigkeit = 0; // in Knoten
    private int ankerAnzahl;
    private boolean satellitenNavigation = true;
    private boolean radar = true;

// Klassenvariablen
    private static int anzahl = 0; // Bootsanzahl

// Instanzmethoden
    public void beschleunigen( int delta){ ... }
    public void bremsen( int delta){ ... }
    public void ankern(){ ... }

// Klassenmethoden
    public static void incAnzahl (){ ... }
}
```

Da Klassen *Referenzdatentypen* sind, werden die Objekte einer Klasse, in zwei Schritten vereinbart:

Instanziierung

```
MotorBoot prinzess, karibik, rostock;

prinzess = new MotorBoot();
karibik  = new MotorBoot();
rostock  = new MotorBoot();
```

oder auch

```
MotorBoot prinzess = new MotorBoot();
MotorBoot karibik  = new MotorBoot();
MotorBoot rostock  = new MotorBoot();
```

Welches Objekt gehört zu welcher Klasse?

Mit der Methode `getClass()` kann die Klasse eines Objekts ermittelt werden. Die Methode `getName()` erfragt den Klassennamen einer Klasse.

Die folgende Methode bestimmt den Klassennamen eines Objekts:

```
public String getTyp()  
{  
    return getClass().getName();  
}
```

Wird diese Methode in die Klasse `MotorBoot` aufgenommen, so liefert der Aufruf `rostock.getTyp();` den Klassennamen `MotorBoot`.

Durch die Zusammenfassung von Attributen und Methoden innerhalb von Klassen soll einerseits die Gefahr von Seiteneffekten verhindert werden, indem ein direkter Zugriff von außen auf die Daten unterbunden wird.

Andererseits verbessert die Modifikationsfreundlichkeit erheblich, da Methoden zur erlaubten Manipulation der Daten bereitgestellt werden.

8.3 Vererbung, Generalisierung und Spezialisierung

8.3.1 Abgeleitete Klassen - Spezialisierung

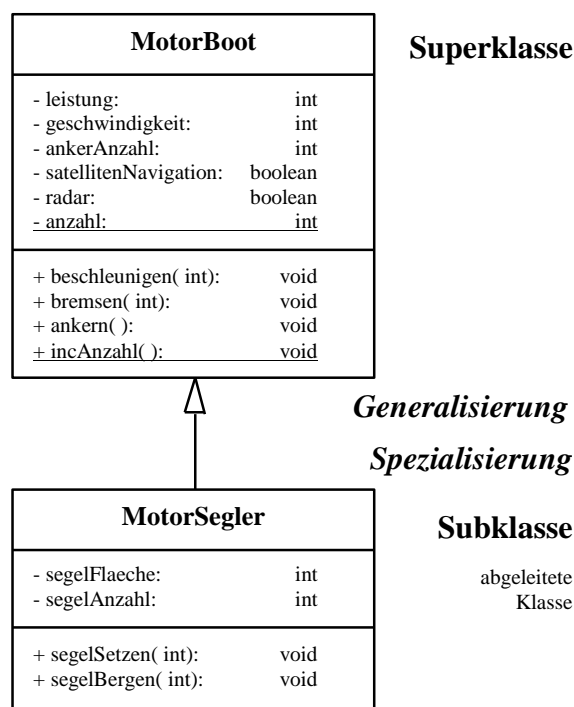
Während man gleichartige Objekte zu einer Klasse zusammenfasst, können ähnliche Objekte *abgeleiteten* Klassen zugeordnet werden.

Dabei besitzt eine von einer **übergeordneten Klasse** (*Superklasse, Oberklasse, Elternklasse*) **abgeleitete Klasse** (*Subklasse, Unterklasse, Kindklasse*) neben den Methoden und Attributen der Superklasse noch einige zusätzliche Attribute oder Methoden.

Deklaration einer abgeleiteten Klasse

```
public class Subklasse extends Superklasse { Deklaration ... }
```

Deklaration der von der Klasse `MotorBoot` abgeleiteten Klasse `MotorSegler`



MotorSegler.java

```
public class MotorSegler extends MotorBoot
{
    private int segelFläche; // in qm
    private int segelAnzahl;

    public void segelSetzen( int qm){ ... }
    public void segelBergen( int qm){ ... }
}
```

Instanziierung

```
MotorSegler daphne = new MotorSegler();
```

Ein Objekt der Klasse `MotorSegler` besitzt alle Attribute der Klasse `MotorBoot` und kann Methoden dieser Klasse ausführen.

| MotorSegler Daphne: | MotorSegler | |
|------------------------|-------------|-----------------------|
| - leistung | 100 | ererbte Eigenschaften |
| - geschwindigkeit | 5 kn | |
| - ankerAnzahl | 1 | |
| - satellitenNavigation | ja | |
| - radar | ja | |
| - <u>anzahl</u> | 4 | |
| <hr/> | | |
| - segelFlaeche: | 45 qm | neue Eigenschaften |
| - segelAnzahl: | 2 | |
| <hr/> | | |
| + beschleunigen | | ererbte Methoden |
| + bremsen | | |
| + ankern | | |
| + <u>incAnzahl</u> | | |
| <hr/> | | |
| + segelSetzen | | neue Methoden |
| + segelBergen | | |

Vererbung ist das Konzept der objektorientierten Programmierung und deshalb in allen objektorientierten Sprachen enthalten. Sein Hauptzweck liegt in der Nutzung der Ähnlichkeit von neu zu schaffenden Klassen mit bereits vorhandenen Klassen auszunutzen, und zwar im Sinne von

- **Spezialisierung**

Erweiterung - spezifische Attribute und Methoden legt man in einer *Subklasse* an und

- **Generalisierung**

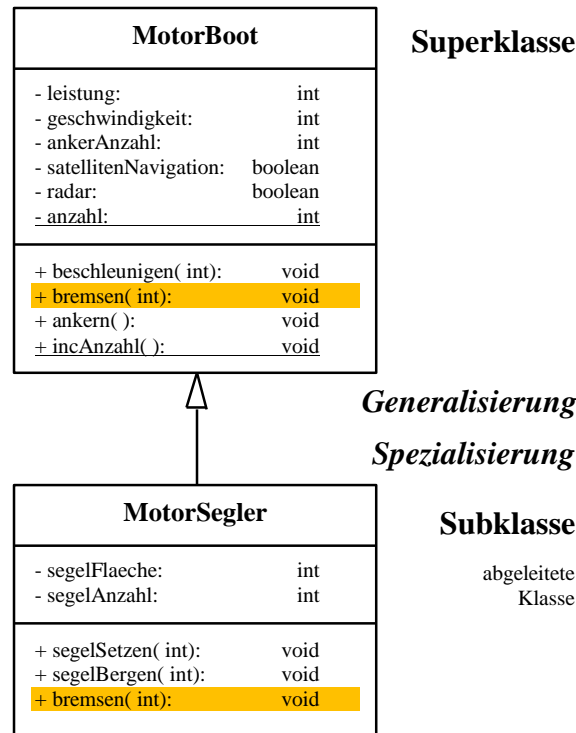
Abstraktion - allgemeingültige Attribute und Methoden gehören in eine *Superklasse*.

Durch Vererbung kann man sich Entwicklungsarbeit sparen. Die verwendeten Klassen sind bereits ausgetestet. Programmierfehler können dann nur noch in den Erweiterungen auftauchen.

8.3.2 Überschreiben von Methoden

Außer dem *Überladen von Methoden* (s.o.) besteht innerhalb einer Klassenhierarchie die Möglichkeit des **Überschreibens von Methoden**. Innerhalb einer Subklasse wird eine komplett neue Methode mit *identischer Schnittstelle* zu einer bereits vorhandenen Methode der Superklasse implementiert. Objekte der Superklasse rufen die Methode der Superklasse auf, während Objekte der Subklasse selbst und aller von der Subklasse abgeleiteten Klassen die neue Methode verwenden.

So ist es denkbar, dass ein Motorsegler beim Bremsen zusätzlich die Segel einsetzt. Die Methode `bremsen` muss deshalb aktualisiert werden. Sie wird *überschrieben*, indem sie in der Klasse `Motorsegler` *neu* vereinbart wird.



8.3.3 Abstrakte Klassen - Generalisierung

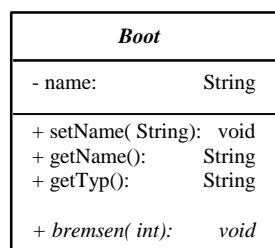
Als *Wurzelklasse* einer Klassenhierarchie verwendet man oft eine **abstrakte Klasse**, die gemeinsame Attribute und Methoden aller *Subklassen* zusammenfasst. Abstrakte Klassen sind *nicht* instanzierbar, d.h. von ihnen lassen sich keine Objekte bilden.

Betrachten wir eine Klasse `Boot` als *Basisklasse* der Bootshierarchie. Alle Boote haben einen Namen. Deshalb wird in dieser Klasse der Name `name` als Attribut angelegt. Da auf `name` *nicht* direkt zugegriffen werden soll, organisieren öffentliche Methoden `setName` und `getName` dieser Klasse das Eintragen und Lesen des Namens. Sie gehören zur *Schnittstelle* der Klasse.

In *abstrakte Klasse* können **abstrakte Methoden** vereinbart werden. Diese sind deklariert, aber noch *nicht* definiert. Abstrakte Methoden *müssen* von allen *direkt* abgeleiteten Klassen überschrieben werden.

Durch Aufnahme einer *abstrakten* Methode `bremsen` in die Klasse `Boot` wird erzwungen, dass jedem abgeleiteten Bootstyp eine spezifische Methode `bremsen` zur Verfügung stehen muss:

```
public abstract void bremsen( int delta);
```



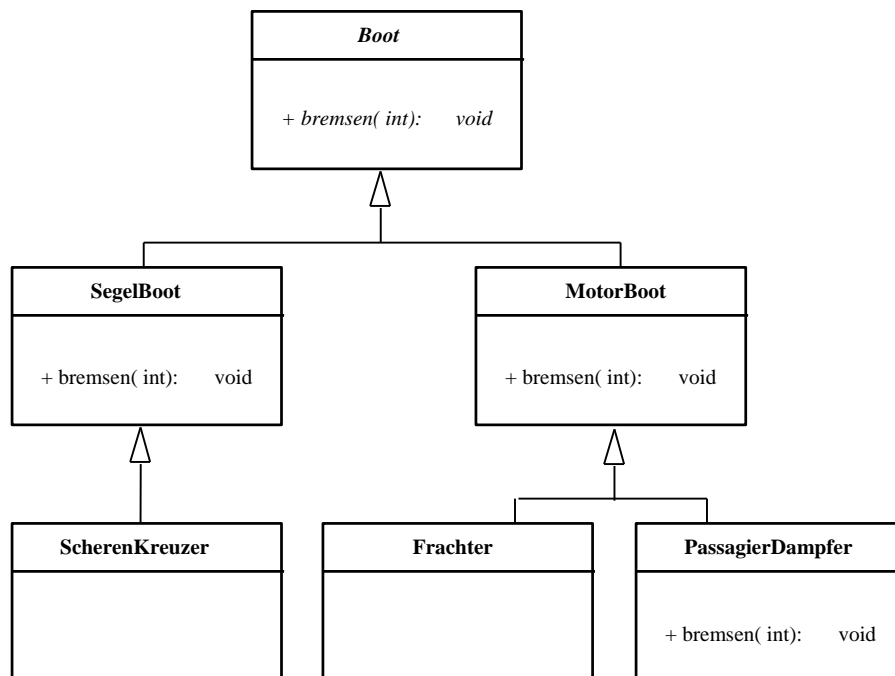
Boot.java

```
// Boot.java
// Boot.java
// MM 2005

/**
 * Boot, Beispiel fuer abstrakte Klassen
 */
public abstract class Boot
{
    private String name;           // Instanzvariable

    public void setName( String name) // Instanzmethoden
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }
    public String getTyp()
    {
        return getClass().getName();
    }
    public abstract void bremsen( int delta);
}

```



Die in der Hierarchie direkt von der Klasse `Boot` abgeleitete Klassen `SegelBoot` bzw. `MotorBoot` *müssen* die Methode `bremsen` definieren. Andere Schiffstypen wiederum *können* das Bremsen durch *Überschreiben* der Methode auf ihren Schiffstyp konkretisieren. Wird dann die Methode `bremsen` aufgerufen, so wird in Abhängigkeit des aufrufenden Objekts entschieden, welches `bremsen` ausgeführt wird.

Objekte der Klassen `SegelBoot` und `ScherenKreuzer` verwenden hier die Methode `bremsen` der Klasse `SegelBoot`, Objekte der Klassen `MotorBoot` und `Frachter` die Methode `bremsen` der Klasse `MotorBoot`. Objekte der Klasse `PassagierDampfer` führen eine eigene Methode `bremsen` aus.

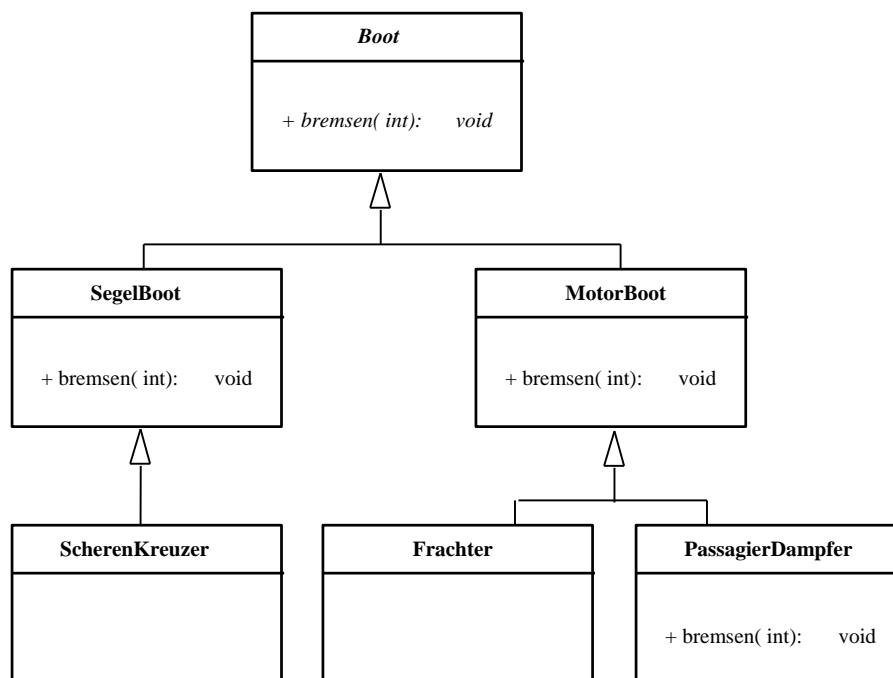
Abstrakte Klassen und abstrakte Methoden werden im Klassendiagramm *kursiv* dargestellt.

8.3.4 Einfach- und Mehrfachvererbung

In Java ist nur einfache Vererbung erlaubt. Mehrfache Vererbung findet man z.B. in C++. Trotzdem soll hier auf diese Problematik eingegangen werden.

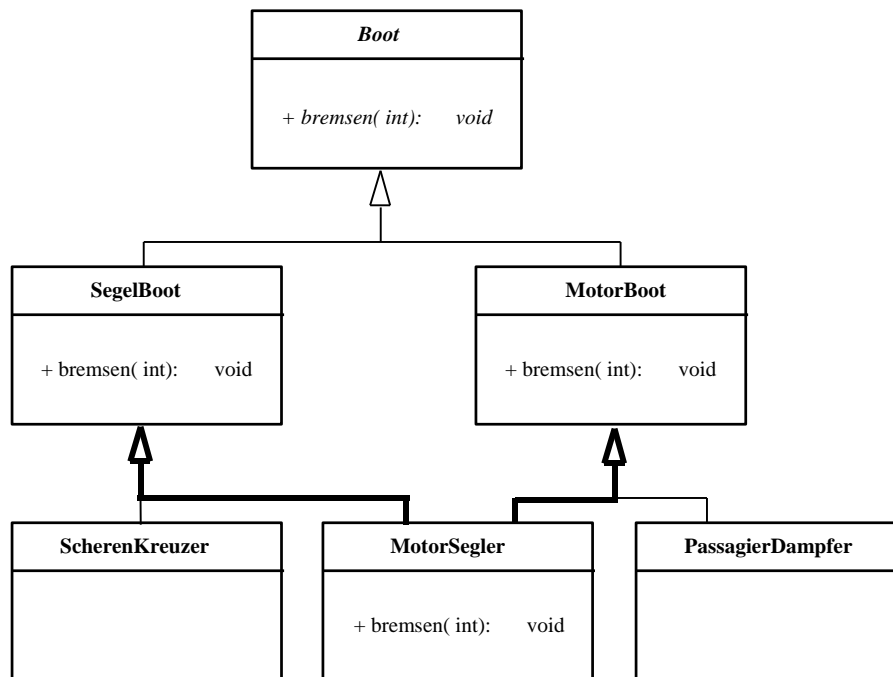
Einfachvererbung

Eine **einfache Vererbung** liegt dann vor, wenn die Vererbungshierarchie ein **Baum** ist, d.h. außer der Basisklasse (**Wurzel**) hat jede Klasse *genau eine direkte Superklasse*.



Mehrfachvererbung

Bei **mehrfacher Vererbung** hat *mindestens eine Klasse mehr als eine direkte Superklasse*. Durch diese Eigenschaft ist es möglich, voneinander unabhängige Klassenhierarchien zusammenzuführen.



Bei der Mehrfachvererbung können **Konflikte** in der Art auftreten, dass Methoden oder Attribute mit dem gleichen Namen in beiden Superklassen unterschiedlich definiert wurden.

Ist die Methode `bremsen` sowohl in der Klasse `SegelBoot` als auch in der Klasse `MotorBoot` definiert, so ist unklar, welches `bremsen` der Klasse `MotorSegler` vererbt wurde. Abhilfe schafft die eigene Realisierung dieser Methode innerhalb der Klasse `MotorSegler`, welche dann die in beiden Superklassen definierte Methode gleichen Namens überschreibt.

8.4 Polymorphismus und dynamisches Binden

Unter **Polymorphismus** (dt. *Vielgestalt*) versteht man die Fähigkeit einer Variablen, auf verschiedene Objekte unterschiedlicher Klassen innerhalb einer Klassenhierarchie zuzugreifen. Polymorphismus hängt eng mit dem *Überschreiben von Methoden* zusammen.

Der Verkehr in einer Hafeneinfahrt soll simuliert werden. Es wurde eine *abstrakte Klasse* (Boot) als Basisklassen eingeführt. Diese Klasse der obersten Ebene (Wurzel des Hierarchiebaumes) definiert u. a. die abstrakte Methode `bremsen`, die von allen Subklassen ererbt wird. Im Hafen befinden sich eine Anzahl von Schiffen, wobei erst zur Laufzeit der jeweilige Bootstyp aller ein- und auslaufenden Schiffe bekannt ist. Erhält ein Kapitän eines Bootes `schiff` nun von der Hafenleitung den Befehl „Bremsen auf 0 Knoten“, so wird der Befehl `schiff.bremsen(0)` auf die Methode des entsprechenden Bootstyp abgebildet, ohne dass der Programmierer bei der Programmentwicklung diesen bereits kennen muss.

Hafen.java

```

...
        // Maximalzahl der im Hafen ankernde Schiffe
final int ANZ_SCHIFFE = 10;
        // Feld von Variablen der Basisklasse
Boot[] schiffe = new Boot[ ANZ_SCHIFFE];

        // Schiffstyp bei Hafeneinfahrt festlegen, z.B.
schiffe[0] = new SegelBoot();
schiffe[1] = new ScherenKreuzer();
schiffe[2] = new Frachter();
...

        // Bremsen aller Schiffe im Hafen
for( int i = 0; i < schiffe.length; i++)
{
    System.out.println
    ( schiffe[ i].getTyp() + " " +
      schiffe[ i].getName() + " bremsen!");
    schiffe[ i].bremsen( 0);    // Methodenaufruf bremsen
}
System.out.println( "Alle Schiffe stehen still!");

```

Hafen.out

```

MotorBoot Prinzess bremsen!
ScherenKreuzer Wacht bremsen!
Frachter Wismar bremsen!
Kutter Klabautermann bremsen!
...

Alle Schiffe stehen still!

```

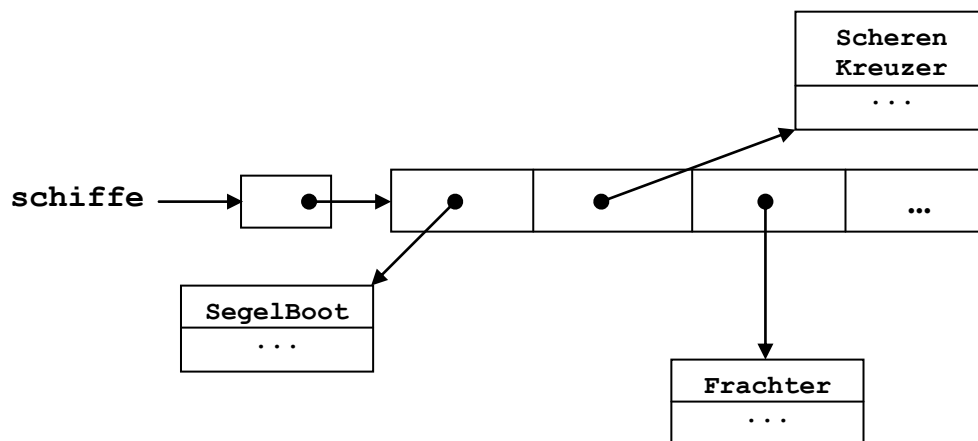
Dem Programmierer wird ermöglicht, Methoden auf einer allgemeinen Klasse zu realisieren.

```
schiffe[ i].bremsen( 0);
```

Zunächst zeigt eine Variable auf eine Basisklasse einer Klassenhierarchie. Zur Laufzeit wird dann spezifiziert, auf welche Methode der Subklassen die Variable tatsächlich zugreift. Gibt es Methoden der Basisklasse, die in der Subklasse überschrieben wurden, so wird der richtige Methodenrumpf zur Laufzeit hinzugebunden (**late binding** - spätes Binden bzw. **dynamisches Binden**). Die interne Verwaltung dieser Methoden erfolgt durch sogenannte **virtuelle Methodentabelle**. Die Auswahl der Methode erfolgt erst zur Laufzeit, bei Smalltalk und Java generell, bei PASCAL und C++ wahlweise.

Mit dieser Technik kann in der Programmierung ein höheres Abstraktionsniveau erreicht werden. Eine vorhandene Klassenhierarchie kann einfach erweitert werden. Neue Unterklassen können hinzugefügt werden, ohne dabei bereits vorhandenen Anwendungsprogramme zu verändern.

Polymorphismus und dynamisches Binden bilden zusammen mit bereits vorhandenen Klassenbibliotheken die Basis für die Wiederverwendbarkeit von Softwarebausteinen.



8.5 Zusammenfassung - Konzepte der OOP

Konzepte der OOP

- **Objekte** sind **Datengrundbausteine** (*Module*), aufgebaut aus **Attributen** (*Daten*) und **Methoden** (*Anweisungen*).
- *Objekte* kommunizieren miteinander über spezielle **öffentliche Methoden** (*Nachrichten*).
- **Klassen** sind Datentypen (*Datenabstraktion*) der Objekte. Sie fassen Objekte mit gleichem Aufbau zusammen und dienen der **Datenkapselung**.
- *Klassen* sind *hierarchisch* aufgebaut. **Vererbungsmechanismen** gestattet die *Weiterentwicklung* neuer Klassen aus bereits existierenden Klassen: Klassen können als **Subklassen** (*Spezialisierung*) abgeleitet und durch **Superklassen** (*Generalisierung*) verallgemeinert werden.
- **Polymorphismus** und **dynamisches Binden** bilden zusammen mit bereits vorhandenen **Klassenbibliotheken** die Basis für die Wiederverwendbarkeit von Softwarebausteinen.

Java Platform, All Classes:

<http://download.oracle.com/javase/6/docs/api/>

Faustregeln zur Modularisierung und Programmentwicklung

- ⇒ **Ein- und Ausgaben** sind im Hinblick auf grafische Oberflächen *nur* in der **main-Methode** oder in speziell dafür vorgesehene Klassen enthalten.
- ⇒ Es sind *kleine Klassen* durch Generalisierung bzw. Spezialisierung zu modellieren, dabei ist auf Wiederverwendbarkeit ist zu achten!
- ⇒ **Jede Klasse, jedes Attribut und jede Methode** einer Klasse sind mit einem externen **Kommentar** `/** */` zu versehen.
- ⇒ **Methoden** sind gut zu strukturieren und sollten *nicht länger* als eine A4-Seite sein. Ansonsten sind sie in kleinere Methoden aufzuteilen. Sie sind mit internen **Kommentaren** `// bzw. /* */` zu versehen!

8.6 Methoden der Klasse `java.lang.String`

Zeichenketten werden in Java *nicht* mittels eines *elementaren Datentyps*, sondern in Form eines *Referenzdatentyps* namens `String` dargestellt. Diese vordefinierte Klasse gehört ebenfalls zum Paket `java.lang` des Standardumfangs von Java.

Alle **Zeichenkettenkonstanten** werden implizit als *Objekt* der Klasse `String` angelegt. Die Werte der *Stringobjekte* können nach ihrer Erzeugung nicht mehr verändert werden. Jede Veränderung an einem `String` durch eine *Stringoperation* liefert ein *neues* `Stringobjekt`.

Es gibt mehrere Möglichkeiten zum **Erzeugen von Stringobjekten**:

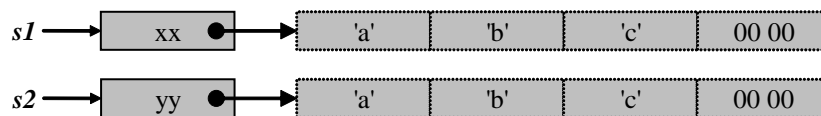
```
String s1 = "abc";
String s2 = new String( "abc");

char[] data = { 'a', 'b', 'c'};
String s3 = new String( data);

byte[] data = { 97, 98, 99};
String s4 = new String( data);

String s5 = new String( s4);
```

`s1` bis `s5` sind Referenzvariablen, die auf unterschiedliche `Stringobjekte` verweisen.



Klassenmethoden

| Name | Parameteranzahl | Parameter-typ | Ergebnis-Typ | Beschreibung |
|----------------|-----------------|---|--------------|----------------------------|
| valueOf | 1 | boolean char char[] double float int long Objekt | String | Umwandlung in einen String |

Der Aufruf einer *Klassenmethode* erfolgt über die Klasse mit dem Punktoperator:

```
String.valueOf( 123)
```

Instanzmethoden (Auswahl)

| Name | Parameteranzahl | Parametertyp | Ergebnistyp | Beschreibung |
|-------------------------------|-----------------|--------------|-------------|--|
| <code>charAt</code> | 1 | int | char | Zeichen an Stelle int |
| <code>compareTo</code> | 1 | String | 1, 0, -1 | lexikografischer Vergleich |
| <code>endsWith</code> | 1 | String | boolean | Vergleich Endung |
| <code>equals</code> | 1 | String | boolean | Vergleich auf Gleichheit |
| <code>equalsIgnoreCase</code> | 1 | String | boolean | Vergleich auf Gleichheit, ohne Groß-/Kleinschreibung |
| <code>indexOf</code> | 1 | String | int | Suche nach Teilstring |
| <code>length</code> | 0 | | int | Stringlänge |
| <code>replace</code> | 2 | char char | String | Einsetzen |
| <code>startsWith</code> | 1 | String | boolean | Vergleich Anfang |
| <code>substring</code> | 1 | int | String | Teilstring ab Stelle int |
| <code>substring</code> | 2 | int int | String | Teilstring von Stelle int bis Stelle int |
| <code>toLowerCase</code> | 0 | | String | Umwandeln in Kleinbuchstaben |
| <code>toUpperCase</code> | 0 | | String | Umwandeln in Großbuchstaben |

Der Aufruf einer Instanzmethode erfolgt über das Objekt mit dem Punktoperator:

```
s1.equals( s2)          =>          true
```

Referenzdatentypen liefern bei einem unmittelbaren Vergleich der Variablen mittels `==` stets den Wert **false**.

```
s1 == s2                =>          false
```

