Einführung in die Programmierung

Vorlesungsteil 6

Objektorientiert programmieren

Klassen, Objekte, Kapselung, Vererbung, Polymorphie

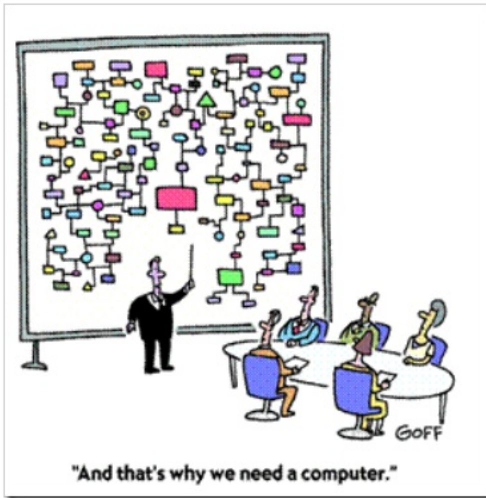
PD Dr. Thomas Hinze

Brandenburgische Technische Universität Cottbus – Senftenberg
Institut für Informatik

Sommersemester 2016

Brandenburgische
Technische Universität
Cottbus - Senftenberg

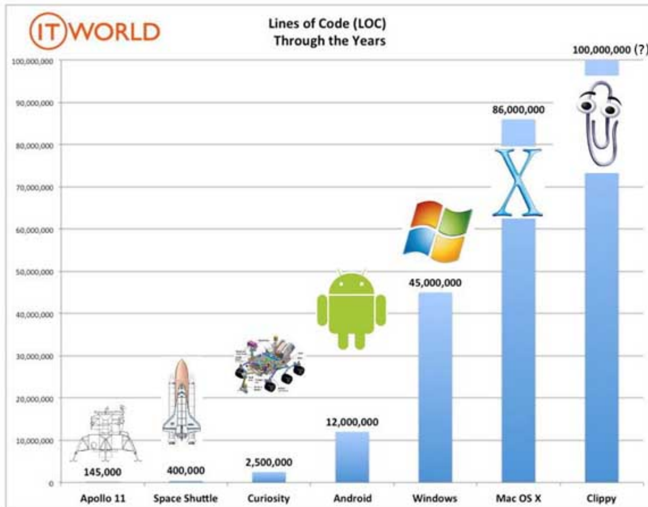
Softwareprojekte werden häufig groß und komplex



www.goff.com

Wachsende Anzahl Quelltextzeilen

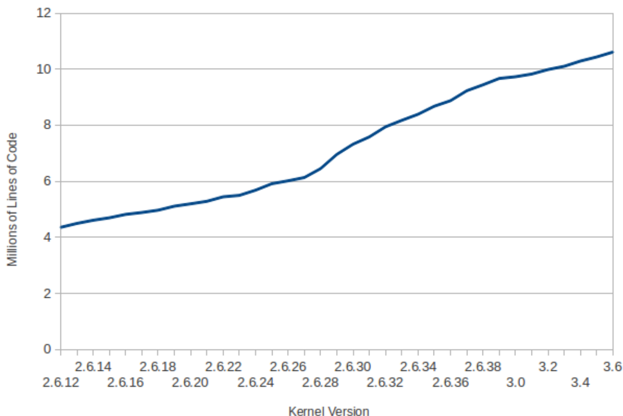
Leistungsfähigere Hardware zieht umfangreichere Software nach sich



www.it-world.com

Mehr Features heißt i.A. mehr Quelltext

Anzahl Quelltextzeilen wächst zumeist auch von Version zu Version



www.dwheeler.com/sloccount/

Objektorientierte Programmierung für große Projekte

- *Softwarekrise* in den 1970er und 80er Jahren, da größer werdende Programmierprojekte immer schlechter beherrscht wurden und dadurch *gehäuft Softwarefehler* auftraten, die *wirtschaftlichen Schaden* verursachten.

Objektorientierte Programmierung für große Projekte

- *Softwarekrise* in den 1970er und 80er Jahren, da größer werdende Programmierprojekte immer schlechter beherrscht wurden und dadurch *gehäuft Softwarefehler* auftraten, die *wirtschaftlichen Schaden* verursachten.
- *Monolithisch* und *modular* aufgebaute Quelltexte uferten aus, *Übersicht* ging leicht *verloren* bei mehreren 100 000 Funktionen, die in großen Projekten erreicht werden. Quelltextanpassungen extrem aufwendig und *fehlerträchtig*. Funktionen häufig unvorteilhaft spezifiziert.

Objektorientierte Programmierung für große Projekte

- *Softwarekrise* in den 1970er und 80er Jahren, da größer werdende Programmierprojekte immer schlechter beherrscht wurden und dadurch *gehäuft Softwarefehler* auftraten, die *wirtschaftlichen Schaden* verursachten.
- *Monolithisch* und *modular* aufgebaute Quelltexte uferten aus, *Übersicht* ging leicht *verloren* bei mehreren 100 000 Funktionen, die in großen Projekten erreicht werden. Quelltextanpassungen extrem aufwendig und *fehlerträchtig*. Funktionen häufig unvorteilhaft spezifiziert.

Objektorientiertes Programmierparadigma begründet Wissenschaftszweig der *Softwaretechnologie* und unterstützt eine recht allgemein anwendbare Vorgehensweise zur Umsetzung und Wartung großer Softwareprojekte.

Idee den Ingenieuren abgeschaut: Autos als Objekte

individualisierbarer Bauplan • Bedienung nur über bereitgestellte Elemente • Kapselung nutzungsirrelevanter Implementierungsdetails

Vorlesung Einführung in die Programmierung mit Java

1. Einführung und erste Schritte . . . .
. . Installation Java-Compiler, ein erstes Programm: HalloWelt, Blick in den Computer

2. Elementare Datentypen, Variablen, Arithmetik, Typecast . . . .
Java als Taschenrechner nutzen, Tastatureingabe → Formelberechnung → Ausgabe

3. Imperative Kontrollstrukturen . . . .
. . . Befehlsfolgen, Verzweigungen, Schleifen und logische Ausdrücke programmieren

4. Methoden selbst programmieren . . . .
. . . . Methoden als wiederverwendbare Funktionen, Werteübernahme und -rückgabe

5. Rekursion . . . .
. . . . selbstaufrufende Funktionen als elegantes algorithmisches Beschreibungsmittel

6. Objektorientiert programmieren . . . .
. . . . Klassen, Objekte, Attribute, Methoden, Sichtbarkeit, Vererbung, Polymorphie

7. Felder und Graphen . . . .
. . . . . effizientes Handling größerer Datenmengen und Beschreibung von Netzwerken

8. Sortieren . . . .
. . . . klassische Sortierverfahren im Überblick, Laufzeit und Speicherplatzbedarf

9. Zeichenketten, Dateiarbeit, Ausnahmen . . . .
. . . Texte analysieren, ver-/entschlüsseln, Dateien lesen/schreiben, Fehler behandeln

10. Dynamische Datenstruktur „Lineare Liste“ . . . .
. . . . unsere selbstprogrammierte kleine Datenbank

11. Ausblick und weiterführende Konzepte . . . .

Die 51 Schlüsselwörter von Java

Java als kompakte Programmiersprache

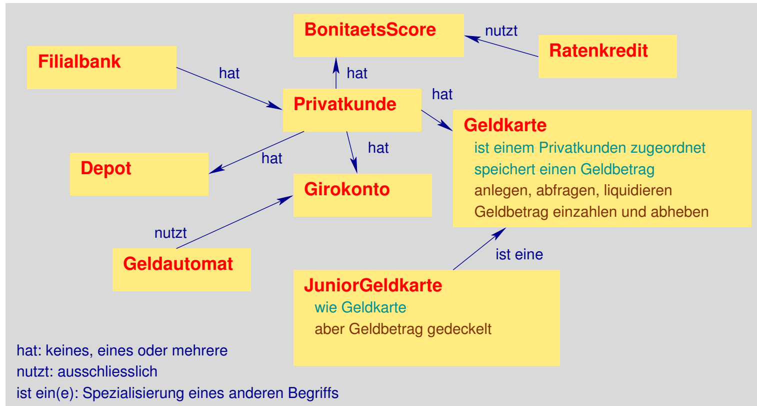
abstract	double	long	static
boolean	else	native	super
break	extends	new	switch
byte	final	null	synchronized
case	finally	operator	this
cast	float	outer	throw
catch	for	package	throws
char	if	private	transient
class	implements	protected	try
const	import	public	var
continue	instanceof	rest	void
default	int	return	while
do	interface	short	

Die 51 Schlüsselwörter von Java

Heute und nächste Woche lernen wir davon kennen . . .

`abstract`	`double`	`long`	`static`
`boolean`	`else`	`native`	**`super`**
`break`	**`extends`**	**`new`**	`switch`
`byte`	**`final`**	**`null`**	`synchronized`
`case`	`finally`	`operator`	**`this`**
`cast`	`float`	`outer`	`throw`
`catch`	`for`	**`package`**	`throws`
`char`	`if`	**`private`**	`transient`
`class`	`implements`	**`protected`**	`try`
`const`	`import`	**`public`**	`var`
`continue`	`instanceof`	`rest`	`void`
`default`	`int`	`return`	`while`
`do`	`interface`	`short`	

Identifizierung grundlegender Projekt-Begriffe



Große Softwareprojekte beginnt man typischerweise damit, dass *grundlegende Begriffe* definiert werden mit ihren *Features* und *Eigenschaften*. Darüber hinaus werden *Beziehungen* und *Abhängigkeiten* zwischen diesen Begriffen charakterisiert. Beispiel: fiktive Bankensoftware

Geldkarte als Klasse

Eigenschaften (Zustandsdaten)

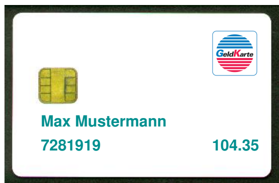
Name des Privatkunden, Geldkartennummer,
hinterlegter Geldbetrag in Euro

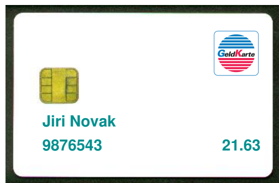
Features (Methoden)

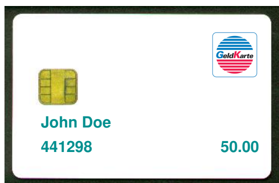
anlegen, abfragen, liquidieren, einzahlen, auszahlen

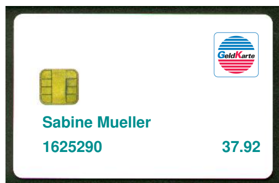
Klasse: Beschreibungsrahmen („individualisierbarer Bauplan“), der *Zustandsdaten* und *darauf operierende Methoden* aufnimmt

Verschiedene Objekte der Klasse Geldkarte









Objekte: individuelle Ausprägungen (Exemplare) der Klasse, in denen die Zustandsdaten jeweils mit *konkreten Werten* belegt sind

Definitionen Klasse, Objekt, Attribut

Eine **Klasse** ist in der objektorientierten Programmierung ein Beschreibungsrahmen (eine Spezifikation), der Zustandsdaten (Attribute) und darauf operierende Funktionen (Methoden) aufnimmt.

Ein **Objekt** ist ein Exemplar (eine Ausprägung, eine Instanz, ein Individuum) einer Klasse, bei dem die Zustandsdaten (Attribute) mit konkreten Werten belegt sind.

Ein **Attribut** (eine Eigenschaft) ist ein granulares Merkmal der Zustandsdaten.

Merke

- Eine *Klasse* kann man als *Verallgemeinerung eines Typs* auffassen, der neben Werten auch Werkzeuge enthält.

Merke

- Eine *Klasse* kann man als *Verallgemeinerung eines Typs* auffassen, der neben Werten auch Werkzeuge enthält.
- *Zu einer Klasse* kann es *beliebig viele Objekte* geben, die sich wie Individuen durch eigene Wertebelegungen der Zustandsdaten (Attribute) unterscheiden.

Merke

- Eine *Klasse* kann man als *Verallgemeinerung eines Typs* auffassen, der neben Werten auch Werkzeuge enthält.
- *Zu einer Klasse* kann es *beliebig viele Objekte* geben, die sich wie Individuen durch eigene Wertebelegungen der Zustandsdaten (Attribute) unterscheiden.
- Ein *Objekt* kann man als *Verallgemeinerung einer Variable* auffassen, die die Zustandsdaten enthält und zudem den Zugang zu den hinterlegten Methoden ermöglicht.

Merke

- Eine *Klasse* kann man als *Verallgemeinerung eines Typs* auffassen, der neben Werten auch Werkzeuge enthält.
- *Zu einer Klasse* kann es *beliebig viele Objekte* geben, die sich wie Individuen durch eigene Wertebelegungen der Zustandsdaten (Attribute) unterscheiden.
- Ein *Objekt* kann man als *Verallgemeinerung einer Variable* auffassen, die die Zustandsdaten enthält und zudem den Zugang zu den hinterlegten Methoden ermöglicht.
- Die in den Attributen gespeicherten Zustandsdaten *überdauern* den Aufruf der in der Klasse verfügbaren Methoden.

Merke

- Eine *Klasse* kann man als *Verallgemeinerung eines Typs* auffassen, der neben Werten auch Werkzeuge enthält.
- *Zu einer Klasse* kann es *beliebig viele Objekte* geben, die sich wie Individuen durch eigene Wertebelegungen der Zustandsdaten (Attribute) unterscheiden.
- Ein *Objekt* kann man als *Verallgemeinerung einer Variable* auffassen, die die Zustandsdaten enthält und zudem den Zugang zu den hinterlegten Methoden ermöglicht.
- Die in den Attributen gespeicherten Zustandsdaten *überdauern* den Aufruf der in der Klasse verfügbaren Methoden.
- Auf Objekte darf ausschließlich mit den darauf hinterlegten Methoden operiert werden (*Kapselung*). Dadurch wird eine definierte Schnittstelle bereitgestellt, die die Objekte vor unerwünschten Manipulationen schützt.

Klasse Geldkarte Schritt für Schritt programmiert

```
public class Geldkarte {

  // Attribute

  private String kundenname;
  private long geldkartennummer;
  private double geldbetrag;

  // Methoden

  //... werden hier eingefuegt

}
```

- Wir legen die Klasse `Geldkarte` an und
- schreiben zunächst die *Attribute* hinein. Diese werden wie gewöhnliche Variablen deklariert, denen aber noch keine Werte zugewiesen werden.
- Das Schlüsselwort `private` stellt sicher, dass nur *innerhalb* der Klasse `Geldkarte` auf die Attributwerte lesend und schreibend zugegriffen werden kann. Man kann die Werte von außen nicht direkt auslesen oder gar verändern. Wir zwingen den Nutzer, dafür die Methoden zu verwenden, die wir ihm gleich zur Verfügung stellen.

Konstruktoren hinzufügen

```
public class Geldkarte {

  // Attribute

  private String kundenname;
  private long geldkartennummer;
  private double geldbetrag;

  // Methoden

  // Konstruktoren-Methoden

  Geldkarte(String kundenname, long geldkartennummer) {
    this.kundenname = kundenname;
    this.geldkartennummer = geldkartennummer;
    this.geldbetrag = 0.0;
  }

  Geldkarte(String kundenname, long geldkartennummer, double geldbetrag) {
    this.kundenname = kundenname;
    this.geldkartennummer = geldkartennummer;
    if (geldbetrag >= 0) {
      this.geldbetrag = geldbetrag;
    } else {
      this.geldbetrag = 0.0;
    }
  }
```

Konstruktoren sind Methoden, mit denen beim Anlegen eines neuen Objektes die Attributwerte initialisiert werden

Merke

- Ein *Konstruktor* ist eine Methode *ohne Rückgabetyp*, die *genauso heißt wie die Klasse*.

Merke

- Ein *Konstruktor* ist eine Methode *ohne Rückgabetyp*, die *genauso heißt wie die Klasse*.
- Einem Konstruktor kann man *Variablenwerte übergeben*, mit denen üblicherweise die Attribute belegt (initialisiert) werden.

Merke

- Ein *Konstruktor* ist eine Methode *ohne Rückgabetyp*, die *genauso heißt wie die Klasse*.
- Einem Konstruktor kann man *Variablenwerte übergeben*, mit denen üblicherweise die Attribute belegt (initialisiert) werden.
- Durch die Möglichkeit, Methoden zu *überladen*, lassen sich *mehrere Konstruktoren* für dieselbe Klasse definieren.

Merke

- Ein *Konstruktor* ist eine Methode *ohne Rückgabetyp*, die *genauso heißt wie die Klasse*.
- Einem Konstruktor kann man *Variablenwerte übergeben*, mit denen üblicherweise die Attribute belegt (initialisiert) werden.
- Durch die Möglichkeit, Methoden zu *überladen*, lassen sich *mehrere Konstruktoren* für dieselbe Klasse definieren.
- Innerhalb der Konstruktoren (und aller weiteren Methoden der Klasse) greift man mithilfe des Schlüsselwortes **`this`** gefolgt vom Punktoperator **`.`** auf die Attribute zu.
  Gibt es keine Namenskonflikte (gleichnamige lokale Variablen), darf man zum Zugriff auf die Attribute innerhalb der gleichen Klasse das **`this.`** auch weglassen.

Merke

- Ein *Konstruktor* ist eine Methode *ohne Rückgabetyp*, die *genauso heißt wie die Klasse*.
- Einem Konstruktor kann man *Variablenwerte übergeben*, mit denen üblicherweise die Attribute belegt (initialisiert) werden.
- Durch die Möglichkeit, Methoden zu *überladen*, lassen sich *mehrere Konstruktoren* für dieselbe Klasse definieren.
- Innerhalb der Konstruktoren (und aller weiteren Methoden der Klasse) greift man mithilfe des Schlüsselwortes **`this`** gefolgt vom Punktoperator **`.`** auf die Attribute zu.
  Gibt es keine Namenskonflikte (gleichnamige lokale Variablen), darf man zum Zugriff auf die Attribute innerhalb der gleichen Klasse das **`this.`** auch weglassen.
- Zu jeder Klasse definiert Java automatisch einen *Standardkonstruktor*, der keine Werte entgegennimmt und auch sonst nichts tut. In unserem Beispiel hat der Standardkonstruktor den Methodenkopf: **`Geldkarte()`**

Methoden zum Auslesen von Attributwerten

```java
// Methoden zum Auslesen der Attributwerte ("getter")

public String gibKundenname() {
  return this.kundenname;
}

public long gibGeldkartennummer() {
  return this.geldkartennummer;
}

public double gibGeldbetrag() {
  return this.geldbetrag;
}
```

- Wir zwingen den Nutzer der Klasse, zum *Auslesen* von Attributwerten genau die Methoden zu verwenden, die wir ihm dazu bereitstellen.

Methoden zum Auslesen von Attributwerten

```
// Methoden zum Auslesen der Attributwerte ("getter")

public String gibKundenname() {
  return this.kundenname;
}

public long gibGeldkartennummer() {
  return this.geldkartennummer;
}

public double gibGeldbetrag() {
  return this.geldbetrag;
}
```

- Wir zwingen den Nutzer der Klasse, zum *Auslesen* von Attributwerten genau die Methoden zu verwenden, die wir ihm dazu bereitstellen.
- Dadurch behalten wir die Hoheit über unsere Implementierung und könnten bspw. klassenintern Veränderungen vornehmen (z.B. das Attribut für die Geldkartennummer als **`String`** statt als **`long`** speichern), ohne dass der Nutzer davon berührt wird.

Methoden zum Auslesen von Attributwerten

```
// Methoden zum Auslesen der Attributwerte ("getter")

public String gibKundenname() {
  return this.kundenname;
}

public long gibGeldkartennummer() {
  return this.geldkartennummer;
}

public double gibGeldbetrag() {
  return this.geldbetrag;
}
```

- Wir zwingen den Nutzer der Klasse, zum *Auslesen* von Attributwerten genau die Methoden zu verwenden, die wir ihm dazu bereitstellen.
- Dadurch behalten wir die Hoheit über unsere Implementierung und könnten bspw. klassenintern Veränderungen vornehmen (z.B. das Attribut für die Geldkartennummer als **`String`** statt als **`long`** speichern), ohne dass der Nutzer davon berührt wird.
- Methoden, die Attributwerte nach außen geben, heißen *„getter“*.

Methoden zum Auslesen von Attributwerten

```
// Methoden zum Auslesen der Attributwerte ("getter")

public String gibKundenname() {
  return this.kundenname;
}

public long gibGeldkartennummer() {
  return this.geldkartennummer;
}

public double gibGeldbetrag() {
  return this.geldbetrag;
}
```

- Wir zwingen den Nutzer der Klasse, zum *Auslesen* von Attributwerten genau die Methoden zu verwenden, die wir ihm dazu bereitstellen.
- Dadurch behalten wir die Hoheit über unsere Implementierung und könnten bspw. klassenintern Veränderungen vornehmen (z.B. das Attribut für die Geldkartennummer als **`String`** statt als **`long`** speichern), ohne dass der Nutzer davon berührt wird.
- Methoden, die Attributwerte nach außen geben, heißen *„getter“*.
- Da diese Methoden von außerhalb der Klasse aufrufbar sein sollen, werden sie als **`public`** gekennzeichnet.

Methoden zum Verändern von Attributwerten

```
// Methoden zum Veraendern der Attributwerte ("setter")

public boolean einzahlen(double x) {
  if (x > 0.0) {
    this.geldbetrag = this.geldbetrag + x;
    return true;
  }
  return false;
}

public boolean auszahlen(double x) {
  if ((x > 0.0) && (this.geldbetrag >= x)) {
    this.geldbetrag = this.geldbetrag - x;
    return true;
  }
  return false;
}

} // class
```

- Die Methoden `einzahlen` und `auszahlen` schreiben der Geldkarte einen Geldbetrag **x** gut bzw. ziehen den Geldbetrag **x** ab.

Methoden zum Verändern von Attributwerten

```
  // Methoden zum Veraendern der Attributwerte ("setter")

  public boolean einzahlen(double x) {
    if (x > 0.0) {
      this.geldbetrag = this.geldbetrag + x;
      return true;
    }
    return false;
  }

  public boolean auszahlen(double x) {
    if ((x > 0.0) && (this.geldbetrag >= x)) {
      this.geldbetrag = this.geldbetrag - x;
      return true;
    }
    return false;
  }

} // class
```

- Die Methoden **`einzahlen`** und **`auszahlen`** schreiben der Geldkarte einen Geldbetrag **`x`** gut bzw. ziehen den Geldbetrag **`x`** ab.
- Im Zuge der Ausführung dieser Methoden kann sich der Attributwert **`geldbetrag`** ändern.

Methoden zum Verändern von Attributwerten

```java
// Methoden zum Veraendern der Attributwerte ("setter")

public boolean einzahlen(double x) {
  if (x > 0.0) {
    this.geldbetrag = this.geldbetrag + x;
    return true;
  }
  return false;
}

public boolean auszahlen(double x) {
  if ((x > 0.0) && (this.geldbetrag >= x)) {
    this.geldbetrag = this.geldbetrag - x;
    return true;
  }
  return false;
}

} // class
```

- Methoden, die Attributwerte ändern können, heißen *„setter“*.

Methoden zum Verändern von Attributwerten

```
// Methoden zum Veraendern der Attributwerte ("setter")

public boolean einzahlen(double x) {
  if (x > 0.0) {
    this.geldbetrag = this.geldbetrag + x;
    return true;
  }
  return false;
}

public boolean auszahlen(double x) {
  if ((x > 0.0) && (this.geldbetrag >= x)) {
    this.geldbetrag = this.geldbetrag - x;
    return true;
  }
  return false;
}

} // class
```

- Methoden, die Attributwerte ändern können, heißen *„setter“*.
- Zusätzlich zu den *settern* kann es in einer Klasse auch weitere Hilfsmethoden geben, die z.B. Daten aufbereiten, Berechnungen dazu ausführen u.ä. Je nachdem, welche Methoden von außen nutzbar sein sollen, markiert man sie entweder als **`public`** oder **`private`**.

Die gesamte Klasse **Geldkarte**

```java
public class Geldkarte {

  // Attribute

  private String kundenname;
  private long geldkartennummer;
  private double geldbetrag;

  // Methoden

  // Konstruktoren-Methoden

  Geldkarte(String kundenname, long geldkartennummer) {
    this.kundenname = kundenname;
    this.geldkartennummer = geldkartennummer;
    this.geldbetrag = 0.0;
  }

  Geldkarte(String kundenname, long geldkartennummer, double geldbetrag) {
    this.kundenname = kundenname;
    this.geldkartennummer = geldkartennummer;
    if (geldbetrag >= 0) {
      this.geldbetrag = geldbetrag;
    } else {
      this.geldbetrag = 0.0;
    }
  }

  // Methoden zum Auslesen der Attributwerte ("getter")

  public String gibKundenname() {
    return this.kundenname;
  }

  public long gibGeldkartennummer() {
    return this.geldkartennummer;
  }

  public double gibGeldbetrag() {
    return this.geldbetrag;
  }

  // Methoden zum Veraendern der Attributwerte ("setter")

  public boolean einzahlen(double x) {
    if (x > 0.0) {
      this.geldbetrag = this.geldbetrag + x;
      return true;
    }
    return false;
  }

  public boolean auszahlen(double x) {
    if ((x > 0.0) && (this.geldbetrag >= x)) {
      this.geldbetrag = this.geldbetrag - x;
      return true;
    }
    return false;
  }

} // class
```

Die Klasse **`Geldkarte`** ist jetzt in Java geschrieben und als Datei **`Geldkarte.java`** abgelegt. Nun wollen wir Objekte davon anlegen und damit arbeiten.

Objekte einer Klasse anlegen

```
public class MeineGeldkartenVerwaltung {

  public static void main(String[] args) {

    // vier neue Geldkarten als Objekte der Klasse Geldkarte anlegen
    Geldkarte karteMustermann = new Geldkarte("Max Mustermann", 7281919L, 100.00);
    Geldkarte karteNovak = new Geldkarte("Jiri Novak", 9876543L);
    Geldkarte karteDoe = new Geldkarte("John Doe", 441298L);
    Geldkarte karteMueller = new Geldkarte("Sabine Mueller", 1625290L, 50.00);
```

- Das Anlegen von Objekten einer Klasse geschieht immer von einer *anderen* Klasse aus. Wir schreiben dazu die Klasse `MeineGeldkartenVerwaltung` und legen die Java-Quelltextdatei ins gleiche Verzeichnis wie `Geldkarte.java`.

Objekte einer Klasse anlegen

```
public class MeineGeldkartenVerwaltung {

  public static void main(String[] args) {

    // vier neue Geldkarten als Objekte der Klasse Geldkarte anlegen
    Geldkarte karteMustermann = new Geldkarte("Max Mustermann", 7281919L, 100.00);
    Geldkarte karteNovak = new Geldkarte("Jiri Novak", 9876543L);
    Geldkarte karteDoe = new Geldkarte("John Doe", 441298L);
    Geldkarte karteMueller = new Geldkarte("Sabine Mueller", 1625290L, 50.00);
```

- Das Anlegen von Objekten einer Klasse geschieht immer von einer *anderen* Klasse aus. Wir schreiben dazu die Klasse `MeineGeldkartenVerwaltung` und legen die Java-Quelltextdatei ins gleiche Verzeichnis wie `Geldkarte.java`.
- Das Anlegen eines Objektes ähnelt dem Anlegen einer Variablen: Zuerst steht der *Klassenname*, dann ein selbstgewählter *Objektname*, gefolgt von **=**.

Objekte einer Klasse anlegen

```
public class MeineGeldkartenVerwaltung {

  public static void main(String[] args) {

    // vier neue Geldkarten als Objekte der Klasse Geldkarte anlegen
    Geldkarte karteMustermann = new Geldkarte("Max Mustermann", 7281919L, 100.00);
    Geldkarte karteNovak = new Geldkarte("Jiri Novak", 9876543L);
    Geldkarte karteDoe = new Geldkarte("John Doe", 441298L);
    Geldkarte karteMueller = new Geldkarte("Sabine Mueller", 1625290L, 50.00);
```

- Das Anlegen von Objekten einer Klasse geschieht immer von einer *anderen* Klasse aus. Wir schreiben dazu die Klasse `MeineGeldkartenVerwaltung` und legen die Java-Quelltextdatei ins gleiche Verzeichnis wie `Geldkarte.java`.
- Das Anlegen eines Objektes ähnelt dem Anlegen einer Variablen: Zuerst steht der *Klassenname*, dann ein selbstgewählter *Objektname*, gefolgt von **=**.
- Das Schlüsselwort **`new`** bewirkt das Anlegen des neuen Objektes im Speicher. Dahinter muss einer der für das neue Objekt verfügbaren *Konstruktoren* aufgerufen werden.

Mit Objekten arbeiten

```
    // mit den Objekten arbeiten
    karteMustermann.einzahlen(4.35);
    karteDoe.einzahlen(50.00);
    karteNovak.einzahlen(40.00);
    karteMueller.auszahlen(12.08);
    karteNovak.auszahlen(20.00);
    karteNovak.einzahlen(1.63);

    System.out.println(karteMustermann.gibKundenname() + "    " + karteMustermann.gibGeldbetrag());
    System.out.println(karteNovak.gibKundenname() + "    " + karteNovak.gibGeldbetrag());
    System.out.println(karteDoe.gibKundenname() + "    " + karteDoe.gibGeldbetrag());
    System.out.println(karteMueller.gibKundenname() + "    " + karteMueller.gibGeldbetrag());
  } //main
} //class
```

- Nachdem ein Objekt angelegt wurde, kann damit gearbeitet werden, indem seine *Methoden aufgerufen* werden.
- Man identifiziert jedes Objekt über seinen *Objektnamen* und kann dann mittels des Punktoperators **.** auf die zugänglichen Methoden zugreifen.

Die gesamte Klasse **MeineGeldkartenVerwaltung**

```java
public class MeineGeldkartenVerwaltung {

  public static void main(String[] args) {

    // vier neue Geldkarten als Objekte der Klasse Geldkarte anlegen
    Geldkarte karteMustermann = new Geldkarte("Max Mustermann", 7281919L, 100.00);
    Geldkarte karteNovak = new Geldkarte("Jiri Novak", 9876543L);
    Geldkarte karteDoe = new Geldkarte("John Doe", 441298L);
    Geldkarte karteMueller = new Geldkarte("Sabine Mueller", 1625290L, 50.00);

    // mit den Objekten arbeiten
    karteMustermann.einzahlen(4.35);
    karteDoe.einzahlen(50.00);
    karteNovak.einzahlen(40.00);
    karteMueller.auszahlen(12.08);
    karteNovak.auszahlen(20.00);
    karteNovak.einzahlen(1.63);

    System.out.println(karteMustermann.gibKundenname() + "    " + karteMustermann.gibGeldbetrag());
    System.out.println(karteNovak.gibKundenname() + "    " + karteNovak.gibGeldbetrag());
    System.out.println(karteDoe.gibKundenname() + "    " + karteDoe.gibGeldbetrag());
    System.out.println(karteMueller.gibKundenname() + "    " + karteMueller.gibGeldbetrag());
  } //main
} //class
```

Die gesamte Klasse MeineGeldkartenVerwaltung

```
public class MeineGeldkartenVerwaltung {

  public static void main(String[] args) {

    // vier neue Geldkarten als Objekte der Klasse Geldkarte anlegen
    Geldkarte karteMustermann = new Geldkarte("Max Mustermann", 7281919L, 100.00);
    Geldkarte karteNovak = new Geldkarte("Jiri Novak", 9876543L);
    Geldkarte karteDoe = new Geldkarte("John Doe", 441298L);
    Geldkarte karteMueller = new Geldkarte("Sabine Mueller", 1625290L, 50.00);

    // mit den Objekten arbeiten
    karteMustermann.einzahlen(4.35);
    karteDoe.einzahlen(50.00);
    karteNovak.einzahlen(40.00);
    karteMueller.auszahlen(12.08);
    karteNovak.auszahlen(20.00);
    karteNovak.einzahlen(1.63);

    System.out.println(karteMustermann.gibKundenname() + "    " + karteMustermann.gibGeldbetrag());
    System.out.println(karteNovak.gibKundenname() + "    " + karteNovak.gibGeldbetrag());
    System.out.println(karteDoe.gibKundenname() + "    " + karteDoe.gibGeldbetrag());
    System.out.println(karteMueller.gibKundenname() + "    " + karteMueller.gibGeldbetrag());
  } //main
} //class
```

`karteMustermann` .... $100.00 + 4.35 = 104.35$
`karteNovak` ... $0.00 + 40.00 - 20.00 + 1.63 = 21.63$
`karteDoe` .... $0.00 + 50.00 = 50.00$
`karteMueller` .... $50.00 - 12.08 = 37.92$

```
Max Mustermann   104.35
Jiri Novak   21.63
John Doe   50.0
Sabine Mueller   37.92
```

Objekte liquidieren und Verweise auf Objekte

```
karteNovak = null;  //Objekt karteNovak wird liquidiert

karteDoe = karteMustermann; //Objekt karteDoe wird liquidiert,
                            //aber karteDoe verweist jetzt
                            //auf Objekt karteMustermann

System.out.print(karteDoe.gibKundenname() + "    ");
System.out.println(karteDoe.gibGeldbetrag());
```

```
Max Mustermann   104.35
```

- Jedes Objekt belegt mit seinen Attributwerten und Methoden einen eigenen (zusammenhängenden) Bereich im Speicher, der solange reserviert bleibt wie das Objekt existiert.
- Wird das Ende eines Blocks erreicht (`}`), so werden alle darin erzeugten Objekte liquidiert und ihr Speicherplatz wieder freigegeben.
- Durch Zuweisen des vordefinierten Wertes **`null`** kann man ein Objekt bei Bedarf vorzeitig liquidieren.
- Zuweisen des Namens eines anderen Objekts derselben Klasse liquidiert das Objekt ebenfalls, und man kann dann über den Objektnamen mit dem zugewiesenen Objekt arbeiten.

Merke

- In einer Klasse, von der *mehrere Objekte* angelegt werden, darf *keine Methode* als **`static`** gekennzeichnet sein.

Merke

- In einer Klasse, von der *mehrere Objekte* angelegt werden, darf *keine Methode* als **`static`** gekennzeichnet sein.
- Ein Objektname verweist (zeigt) auf ein Objekt (im Speicher). Daher nennt man Objektnamen auch *Referenzen* (auf Objekte).

Merke

- In einer Klasse, von der *mehrere Objekte* angelegt werden, darf *keine Methode* als **`static`** gekennzeichnet sein.
- Ein Objektname verweist (zeigt) auf ein Objekt (im Speicher). Daher nennt man Objektnamen auch *Referenzen* (auf Objekte).
- Methoden in Klassen dürfen *überladen* werden. Das ist häufig sinnvoll und benutzerfreundlich.

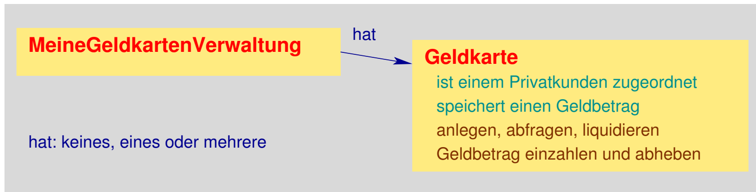
Merke

- In einer Klasse, von der *mehrere Objekte* angelegt werden, darf *keine Methode* als **`static`** gekennzeichnet sein.
- Ein Objektname verweist (zeigt) auf ein Objekt (im Speicher). Daher nennt man Objektnamen auch *Referenzen* (auf Objekte).
- Methoden in Klassen dürfen *überladen* werden. Das ist häufig sinnvoll und benutzerfreundlich.
- Nur solche Methoden als **`public`** kennzeichnen, die von außerhalb der Klasse genutzt werden sollen.

Merke

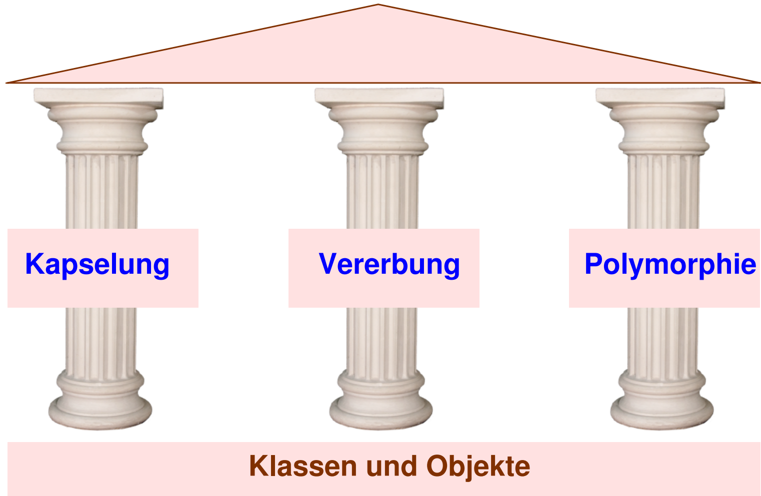
- In einer Klasse, von der *mehrere Objekte* angelegt werden, darf *keine Methode* als **`static`** gekennzeichnet sein.
- Ein Objektname verweist (zeigt) auf ein Objekt (im Speicher). Daher nennt man Objektnamen auch *Referenzen* (auf Objekte).
- Methoden in Klassen dürfen *überladen* werden. Das ist häufig sinnvoll und benutzerfreundlich.
- Nur solche Methoden als **`public`** kennzeichnen, die von außerhalb der Klasse genutzt werden sollen.
- Jede Klasse sollte eine eigenständige Java-Quelltextdatei bilden. Es ist zulässig, mehrere Klassen untereinander in die gleiche java-Datei zuschreiben und diese Datei nach der Klasse zu benennen, in welcher die **`main`**-Methode definiert ist.

Objekte anlegen: hat-Beziehung zwischen Klassen



- Die Klassen **`MeineGeldkartenVerwaltung`** und **`Geldkarte`** stehen miteinander in Beziehung.
- Die Klasse zur Geldkartenverwaltung *hat* (besitzt, verfügt über) Geldkarten (zumindest in gedanklicher Vorstellung).
- Allgemein gilt: Werden in einer Klasse **`A`** Objekte einer Klasse **`B`** angelegt, so stehen beide Klassen in einer *hat-Beziehung*.
- In großen Softwareprojekten mit vielen Klassen besteht ein umfangreiches *Geflecht* aus Beziehungen zwischen den einzelnen Klassen, so dass sich längere Ketten (*Hierarchien*) von hat-Beziehungen ergeben können.

Die drei Säulen der objektorientierten Programmierung



Idee der Kapselung

Einem Pkw-Fahrer werden zur Bedienung (Benutzung) eines Autos bestimmte dafür vorgesehene Bedienelemente zur Verfügung gestellt (Lenkrad, Gaspedal, Bremspedal, Blinker, . . .). Technische Details wie die Interaktion zwischen Gaspedal und Motor bleiben dem Fahrer verborgen und werden vor ihm verkapselt, um unerwünschte Handlungen auszuschließen.

Idee der Kapselung

Einem Pkw-Fahrer werden zur Bedienung (Benutzung) eines Autos bestimmte dafür vorgesehene Bedienelemente zur Verfügung gestellt (Lenkrad, Gaspedal, Bremspedal, Blinker, . . .). Technische Details wie die Interaktion zwischen Gaspedal und Motor bleiben dem Fahrer verborgen und werden vor ihm verkapselt, um unerwünschte Handlungen auszuschließen.

- Der Nutzer einer Klasse darf nur mit den Methoden (Werkzeugen) darauf operieren, die ihm der Programmierer dafür ausdrücklich zur Verfügung stellt.

Idee der Kapselung

Einem Pkw-Fahrer werden zur Bedienung (Benutzung) eines Autos bestimmte dafür vorgesehene Bedienelemente zur Verfügung gestellt (Lenkrad, Gaspedal, Bremspedal, Blinker, . . .). Technische Details wie die Interaktion zwischen Gaspedal und Motor bleiben dem Fahrer verborgen und werden vor ihm verkapselt, um unerwünschte Handlungen auszuschließen.

- Der Nutzer einer Klasse darf nur mit den Methoden (Werkzeugen) darauf operieren, die ihm der Programmierer dafür ausdrücklich zur Verfügung stellt.
- Damit lassen sich Details der Implementierung nach außen verbergen und vor (unbeabsichtigten) Veränderungen oder unerwünschten Bedienhandlungen schützen.

Idee der Kapselung

Einem Pkw-Fahrer werden zur Bedienung (Benutzung) eines Autos bestimmte dafür vorgesehene Bedienelemente zur Verfügung gestellt (Lenkrad, Gaspedal, Bremspedal, Blinker, . . .). Technische Details wie die Interaktion zwischen Gaspedal und Motor bleiben dem Fahrer verborgen und werden vor ihm verkapselt, um unerwünschte Handlungen auszuschließen.

- Der Nutzer einer Klasse darf nur mit den Methoden (Werkzeugen) darauf operieren, die ihm der Programmierer dafür ausdrücklich zur Verfügung stellt.
- Damit lassen sich Details der Implementierung nach außen verbergen und vor (unbeabsichtigten) Veränderungen oder unerwünschten Bedienhandlungen schützen.
- Dadurch lassen sich Programmierfehler in großen Projekten von vornherein minimieren und Aufgaben besser im Team aufteilen.

Realisierung von Kapselung

durch *Sichtbarkeitsvermerke* vor Attributen und Methoden

Staffelung in vier Stufen

Package

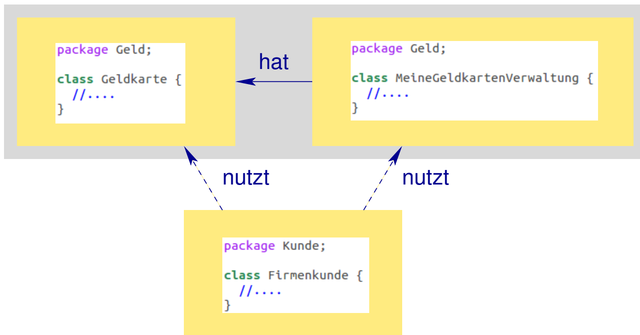
Ein **Package** ist eine Sammlung logisch zusammengehörender Klassen und bildet einen *Namensraum*.

Um eine Klasse als zugehörig zu einem Package zu kennzeichnen, schreibt man *vor* die Klassendefinition und *vor* die `import`-Zeile(n) im Quelltext:

```
package <name>;
```

- Der `<name>` ist ein frei wählbarer Bezeichner. Bei allen Klassen, die zum gleichen Package gehören sollen, muss der gleiche `<name>` stehen.
- Jede Klasse darf zu höchstens einem Package gehören.
- Packages erlauben es, Gruppen von Klassen zu bilden, die in besonders „enger“ Beziehung zueinander stehen.

Packages im Beispiel



Die Klasse `Firmenkunde` darf nur auf solche Attribute und Methoden in den Klassen `Geldkarte` und `MeineGeldkartenVerwaltung` zugreifen, die dort als **`public`** markiert sind.

Die Klasse `MeineGeldkartenVerwaltung` darf auch auf Attribute und Methoden der Klasse `Geldkarte` zugreifen, die *ohne Sichtbarkeitsvermerk* stehen oder als **`protected`** markiert sind.

Staffelung der Sichtbarkeiten

auf Ebene einzelner Attribute und Methoden

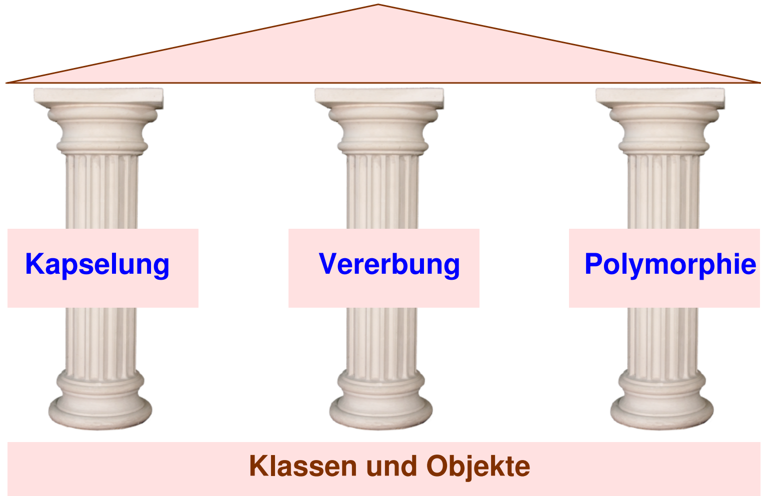
Element ist sichtbar in …	*Element-Modifikator*			
	`public`	`protected`	—	`private`
… der Klasse selbst	✓	✓	✓	✓
… Klassen im gleichen Package	✓	✓	✓	
… Subklassen anderer Packages	✓	✓		
… allen Klassen aller Packages	✓			

auf Ebene ganzer Klassen

Klasse ist sichtbar …	*Klassen-Modifikator*	
	`public`	—
… überall im gleichen Package	✓	✓
… in anderen Packages	✓	

Tabellen aus: P. Pepper, Programmieren lernen, Springer-Verlag, 2010

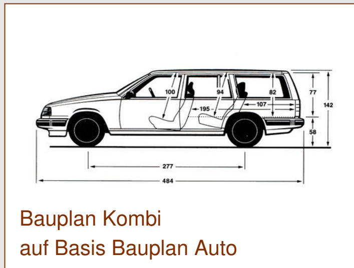
Die drei Säulen der objektorientierten Programmierung



Idee der Vererbung

"Wir wollen Fahrzeuge bauen, die mehr Lasten als Autos transportieren koennen, aber bewaehrte Elemente nachnutzen."

Bauplan Auto



Bauplan Kombi
auf Basis Bauplan Auto

Ein Kombi ist ein spezielles Auto, das aber zusaetzliche Features wie eine Heckklappe besitzt und technisch fuer groessere Lasten ausgelegt ist.

Idee der Vererbung

- *Neue Klassen* auf Basis schon *bestehender Klassen* schreiben

Idee der Vererbung

- *Neue Klassen* auf Basis schon *bestehender Klassen* schreiben
- Die *neue Klasse* *erbt* die *Attribute* und *Methoden* einer anderen Klasse

Idee der Vererbung

- *Neue Klassen* auf Basis schon *bestehender Klassen* schreiben
- Die *neue Klasse* *erbt* die *Attribute* und *Methoden* einer anderen Klasse
- Die ererbten Attribute und Methoden stehen in der neuen Klasse zur Verfügung, ohne dass man sie noch einmal in den Quelltext schreiben muss (sehr bequem und vermeidet fehlerträchtige Quelltext-Duplikate)

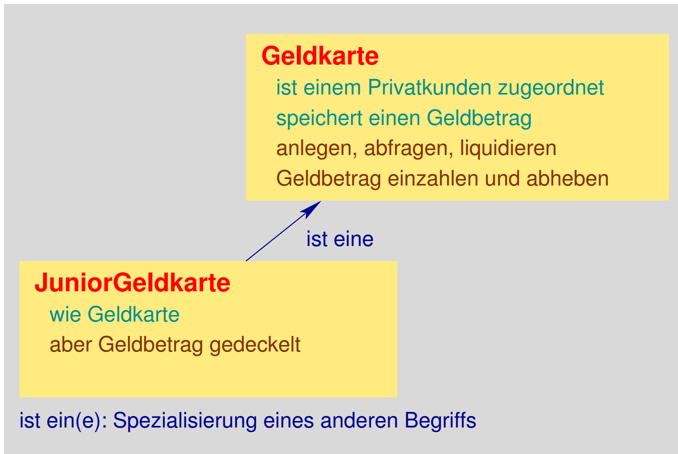
Idee der Vererbung

- *Neue Klassen* auf Basis schon *bestehender Klassen* schreiben
- Die *neue Klasse* *erbt* die *Attribute* und *Methoden* einer anderen Klasse
- Die ererbten Attribute und Methoden stehen in der neuen Klasse zur Verfügung, ohne dass man sie noch einmal in den Quelltext schreiben muss (sehr bequem und vermeidet fehlerträchtige Quelltext-Duplikate)
- In der *erbenden Klasse* wird lediglich notiert, was *neu* dazukommt (weitere Methoden oder Attribute) und was sich *verändert*

Idee der Vererbung

- *Neue Klassen* auf Basis schon *bestehender Klassen* schreiben
- Die *neue Klasse erbt* die *Attribute* und *Methoden* einer anderen Klasse
- Die ererbten Attribute und Methoden stehen in der neuen Klasse zur Verfügung, ohne dass man sie noch einmal in den Quelltext schreiben muss (sehr bequem und vermeidet fehlerträchtige Quelltext-Duplikate)
- In der *erbenden Klasse* wird lediglich notiert, was *neu* dazukommt (weitere Methoden oder Attribute) und was sich *verändert*
- Die *erbende Klasse* ist in ihrer *Funktionalität spezieller* als ihr Vorfahre, aber aus Sicht des Quelltextes *erweitert* sie ihren Vorfahren um zusätzliche Methoden oder Attribute

Vererbung als ist-Beziehung zwischen Klassen



Eine `JuniorGeldkarte` ist eine *spezielle* `Geldkarte`, die *zusätzliche (weitere) Features* besitzt.

Vererbung programmieren

```
public class JuniorGeldkarte extends Geldkarte {
  // zusaetzliche Attribute
  private double maxGuthaben;
  // zusaetzliche Methoden
  // ... hier eintragen
}
```

Vererbung wird durch das Schlüsselwort `extends` initiiert, dahinter steht der Name der Klasse, deren Methoden und Attribute geerbt werden sollen. In Java kann eine Klasse nur von *einer* anderen Klasse erben, mehrere Vorfahren sind nicht zulässig.

Mit `super` auf die übergeordnete Klasse zugreifen

```
public class JuniorGeldkarte extends Geldkarte {

  // zusaetzliche Attribute

  private double maxGuthaben;

  // Konstruktor(en)

  JuniorGeldkarte(String kundenname, long geldkartennummer) {
    super(kundenname, geldkartennummer); //Konstruktor der Klasse Geldkarte
    this.maxGuthaben = 20.00;
  }

  JuniorGeldkarte(String kundenname, long geldkartennummer, double geldbetrag) {
    super(kundenname, geldkartennummer, geldbetrag);
    this.maxGuthaben = 20.00;
  }
```

- Mit dem Schlüsselwort **`super`** auf die Klasse zugreifen, von der geerbt wird (hier: `Geldkarte`)
- In den Konstruktoren der Klasse `JuniorGeldkarte` als erstes die entsprechenden Konstruktoren der Klasse `Geldkarte` aufrufen und übergebene Parameterwerte dorthin weitergeben
- Danach neu hinzugenommene Attribute initialisieren, hier `maxGuthaben` (Maximalbetrag, der auf einer JuniorGeldkarte gespeichert sein kann)

Zusätzliche Methoden einfügen

```
// weitere Methoden
public boolean setzeMaxGuthaben(double x) {
  if (x > 0.0) {
    this.maxGuthaben = x;
    return true;
  }
  return false;
}
```

- In unserem Fall soll eine JuniorGeldkarte eine Methode **`setzeMaxGuthaben`** bereithalten, mit der der maximal auf einer JuniorGeldkarte gespeicherte Geldbetrag festgelegt werden kann (Deckelung des Geldbetrages)
- Alle *nicht* als **`private`** gekennzeichneten Methoden der Klasse **`Geldkarte`** stehen in der Klasse **`JuniorGeldkarte`** zur Verfügung (sind ererbt), ohne dass sie extra in den Quelltext geschrieben werden müssen.

Ererbte Methoden überschreiben

```
// Methode einzahlen ueberschreiben
public boolean einzahlen(double x) {
  if ((x > 0.0) && (this.gibGeldbetrag() + x <= this.maxGuthaben)) {
    super.einzahlen(x); //Methode einzahlen der Klasse Geldkarte aufrufen
    return true;
  }
  return false;
}
```

- Mitunter kommt es vor, dass eine ererbte Methode nicht in ihrer Originalform genutzt werden kann oder soll.
- In unserem Fall betrifft dies die Methode **`einzahlen`**. Die ererbte Methode prüft nämlich beim Einzahlen nicht, ob der maximal zulässige Geldbetrag auf der JuniorGeldkarte überschritten wird.
- Wir lösen dies, indem wir in der Klasse **`JuniorGeldkarte`** einfach die Methode **`einzahlen`** ersetzen durch eine modifizierte Form mit gleicher Argumentliste.
- Mit **`super`** können wir darin bei Bedarf auf die Klasse **`Geldkarte`** zugreifen.

Vererbbare Methoden vor Überschreiben schützen

```
public class Geldkarte {

  //...

  public final boolean einzahlen(double x) {
    if (x > 0.0) {
      this.geldbetrag = this.geldbetrag + x;
      return true;
    }
    return false;
  }
```

Markiert man eine Methode als **`final`**, wird sie zwar vererbt, kann in der erbenden Klasse aber nicht überschrieben werden (ähnlich wie bei einfachen Konstanten, die ebenfalls nicht mit anderen Werten überschrieben werden dürfen).

Typische Einsatzfälle sind sicherheitskritische Methoden wie

- Passwortabfragen
- Methoden zur Ver- und Entschlüsselung
- Methoden zum Generieren und Prüfen digitaler Unterschriften

Die gesamte Klasse **JuniorGeldkarte**

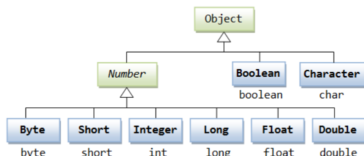
```
public class JuniorGeldkarte extends Geldkarte {

  // zusaetzliche Attribute
  private double maxGuthaben;

  // Konstruktor(en)
  JuniorGeldkarte(String kundenname, long geldkartennummer) {
    super(kundenname, geldkartennummer); //Konstruktor der Klasse Geldkarte
    this.maxGuthaben = 20.00;
  }

  JuniorGeldkarte(String kundenname, long geldkartennummer, double geldbetrag) {
    super(kundenname, geldkartennummer, geldbetrag);
    this.maxGuthaben = 20.00;
  }

  // weitere Methoden
  public boolean setzeMaxGuthaben(double x) {
    if (x > 0.0) {
      this.maxGuthaben = x;
      return true;
    }
    return false;
  }

  // Methode einzahlen ueberschreiben
  public boolean einzahlen(double x) {
    if ((x > 0.0) && (this.gibGeldbetrag() + x <= this.maxGuthaben)) {
      super.einzahlen(x); //Methode einzahlen der Klasse Geldkarte aufrufen
      return true;
    }
    return false;
  }
} //class
```

Die gesamte Klasse MeineGeldkartenVerwaltung

```
public class MeineGeldkartenVerwaltung {

  public static void main(String[] args) {

    // vier neue Geldkarten als Objekte der Klasse Geldkarte anlegen
    Geldkarte karteMustermann = new Geldkarte("Max Mustermann", 7281919L, 100.00);
    Geldkarte karteNovak = new Geldkarte("Jiri Novak", 9876543L);
    Geldkarte karteDoe = new Geldkarte("John Doe", 441298L);
    Geldkarte karteMueller = new Geldkarte("Sabine Mueller", 1625290L, 50.00);

    JuniorGeldkarte kartePaul = new JuniorGeldkarte("Paul Doe", 441299L);

    // mit den Objekten arbeiten
    karteMustermann.einzahlen(4.35);
    karteDoe.einzahlen(50.00);
    karteNovak.einzahlen(40.00);
    karteMueller.auszahlen(12.08);
    karteNovak.auszahlen(20.00);
    karteNovak.einzahlen(1.63);

    kartePaul.setzeMaxGuthaben(50.00);
    kartePaul.einzahlen(10.00);
    kartePaul.auszahlen(4.50); //Klasse JuniorGeldkarte verwendet ererbte Methode auszahlen

    System.out.println(karteMustermann.gibKundenname() + "    " + karteMustermann.gibGeldbetrag());
    System.out.println(karteNovak.gibKundenname() + "    " + karteNovak.gibGeldbetrag());
    System.out.println(karteDoe.gibKundenname() + "    " + karteDoe.gibGeldbetrag());
    System.out.println(karteMueller.gibKundenname() + "    " + karteMueller.gibGeldbetrag());

    System.out.println(kartePaul.gibKundenname() + "    " + kartePaul.gibGeldbetrag());
  } //main
} //class
```

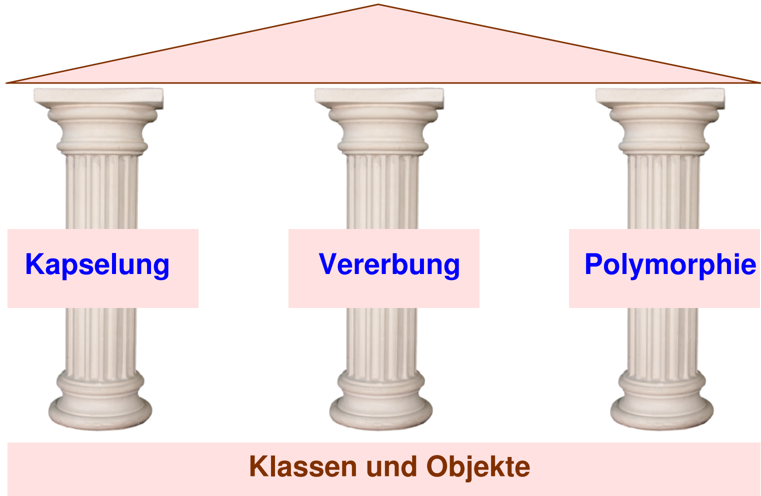
Merke

- Eine Klasse in Java darf nur von *einer* anderen Klasse erben (*Einfachvererbung*), aber nicht von mehreren. Sonst wären Beispiele denkbar, in denen nicht eindeutig ist, welche Methode in die erbende Klasse aufgenommen wird, wenn in den Vorfahren-Klassen gleichnamige Methoden mit gleichen Argumentlisten existieren.
- Es können aber *Vererbungshierarchien* bzw. *Vererbungsbäume* beliebiger Länge bzw. Tiefe auftreten.
- Auch zwischen den schon in Java vordefinierten Klassen bestehen Vererbungsbeziehungen. Die *„Urklasse“*, von der alle anderen erben, heißt: `Object`



Vererbungsbaum von Wrapper-Klassen im Sprachumfang von Java

Die drei Säulen der objektorientierten Programmierung



Idee der Polymorphie

Polymorphie (griech.) bedeutet *Vielgestaltigkeit*

- Ein einfacher Fall von Polymorphie sind überladene Methoden innerhalb einer Klasse (*statische Polymorphie*).

Idee der Polymorphie

Polymorphie (griech.) bedeutet *Vielgestaltigkeit*

- Ein einfacher Fall von Polymorphie sind überladene Methoden innerhalb einer Klasse (*statische Polymorphie*).
- In mehreren Klassen können gleichnamige Methoden mit gleicher Argumentliste definiert sein, deren Implementierungen sich unterscheiden. Überschriebene Methoden in erbenden Klassen sind ein Beispiel dafür. Erst beim konkreten Methodenaufruf erfolgt die Auswahl der zutreffenden Klasse (*dynamische Polymorphie*).

Idee der Polymorphie

Polymorphie (griech.) bedeutet *Vielgestaltigkeit*

- Ein einfacher Fall von Polymorphie sind überladene Methoden innerhalb einer Klasse (*statische Polymorphie*).
- In mehreren Klassen können gleichnamige Methoden mit gleicher Argumentliste definiert sein, deren Implementierungen sich unterscheiden. Überschriebene Methoden in erbenden Klassen sind ein Beispiel dafür. Erst beim konkreten Methodenaufruf erfolgt die Auswahl der zutreffenden Klasse (*dynamische Polymorphie*).
- Darüber hinaus erlaubt es Java, Methoden typübergreifend zu definieren. Zur Implementierung dieses Konzepts, das in der Fachsprache *Generics* oder auch *parametrische Polymorphie* heißt, bedient man sich spezieller *Typvariablen*, die in der Klassendefinition in spitze Klammern gefasst sind, z.B. `<T>`.

Idee der Polymorphie

Polymorphie (griech.) bedeutet *Vielgestaltigkeit*

- Ein einfacher Fall von Polymorphie sind überladene Methoden innerhalb einer Klasse (*statische Polymorphie*).
- In mehreren Klassen können gleichnamige Methoden mit gleicher Argumentliste definiert sein, deren Implementierungen sich unterscheiden. Überschriebene Methoden in erbenden Klassen sind ein Beispiel dafür. Erst beim konkreten Methodenaufruf erfolgt die Auswahl der zutreffenden Klasse (*dynamische Polymorphie*).
- Darüber hinaus erlaubt es Java, Methoden typübergreifend zu definieren. Zur Implementierung dieses Konzepts, das in der Fachsprache *Generics* oder auch *parametrische Polymorphie* heißt, bedient man sich spezieller *Typvariablen*, die in der Klassendefinition in spitze Klammern gefasst sind, z.B. **<T>**.

$\Longrightarrow$ weiterführende Konzepte über EidP hinaus