Strings in Java

- In der GUI wird eine Zeichenkette vom Typ `String` erzeugt.
- Im Programm wird ebenfalls ein `String` erzeugt.
- Selbst wenn beide Zeichenketten aus den gleichen Buchstaben bestehen, sind die beiden Zeichenketten **verschiedene Objekte**.
  - Folglich liefert ein **Referenzvergleich** das Ergebnis `false`.
- Wir wissen natürlich(!?!) aus SE1: Strings sollten **ausschließlich** über die für alle Java-Objekte definierte Operation `equals` miteinander verglichen werden.
- Die entscheidende Frage aber lautet: Wie nützlich ist die Tatsache, dass ich mehrere Zeichenketten aus den gleichen Buchstaben erzeugen und sie trotzdem voneinander unterscheiden kann?
- Meine Antwort lautet: **Völlig unnütz!**
- Strings werden in Java speziell behandelt, aber leider nicht konsequent.

Inkonsequente Spezialbehandlung von Strings in Java

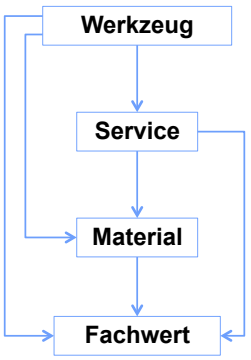
- Für Strings gibt es in der Sprache explizit definierte **Literale**:
  `"göttlich"`
- Für Strings ist der **Operator +** ad hoc überladen:
  `"prächtig" + ", prächtig"` → `"prächtig, prächtig"`
- Außerdem sind String-Objekte unveränderlich (nichts Spezielles hier).

- Es wäre leicht gewesen, in der Sprache eine weitere Sonderregel einzuführen, die ein `==` zweier Strings auf einen Aufruf von `equals` abbildet:
  `s1 == s2` → `s1.equals(s2)`
- String würden sich dann genauso verhalten wie die primitiven Typen `int`, `float` etc.

- Stattdessen baden Generationen von Studierenden diesen Designfehler aus!

Wdh.: Die SE2-Entwurfsregeln

- Die **SE2-Entwurfsregeln** benennen vier **Elementtypen**, aus denen sich ein interaktives System zusammensetzt:
  - **Materialien** realisieren veränderliche, anwendungsfachliche Gegenstände.
  - **Fachwerte** sind anwendungsfachliche Werte; sie sind unveränderlich.
  - **Werkzeuge** bieten eine grafische Benutzungsschnittstelle und ermöglichen das interaktive Bearbeiten von Materialien.
  - **Services** bieten materialübergreifend fachliche Dienstleistungen an, die systemweit zur Verfügung stehen sollen.



Die Pfeile zeigen die **erlaubten Benutzt-Beziehungen** zwischen den Elementtypen. Jeder Elementtyp kann außerdem Elemente vom eigenen Typ benutzen (hier nicht dargestellt).

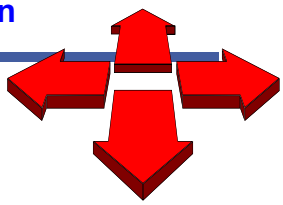
Wdh.: Fachwerte

- Bei der Modellierung eines Anwendungsbereichs gibt es immer auch Begriffe, die eher wertartige Dinge beschreiben, wie **Kontonummer** oder **Geldbetrag**.
- Wir beschreiben solche Begriffe über **Fachwerte**.
- Fachwerte sind fachlich motivierte **Werte**. Werte sind ein allgemeineres Konzept, das beispielsweise auch Zahlen und Zeichenketten umfasst.
  - Ein Wert ist **unveränderlich**.
  - Wir beschreiben Werte programmiertechnisch über **Werttypen**.
  - Werttypen sind besondere Typen mit einer **unveränderlichen Wertemenge**; Werte werden somit konzeptuell nicht erzeugt, sondern bei Bedarf aus der Wertemenge ausgewählt.
- Fachwerte bilden die Grundkonstanten in einem Anwendungssystem.

heute

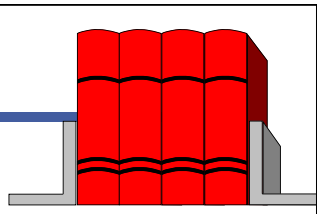
- Wir werden uns Werttypen ~~in einer der nächsten Vorlesungen~~ ausführlich ansehen.

Werte und Objekte in Programmiersprachen



- Warum Werte?
- Wie erkennen wir Werte?
- Wie unterstützen uns Programmiersprachen bei Werten?
- Wie programmieren wir Werttypen in Java?

Literaturhinweise



B.J. **MacLennan**, *Values and Objects in Programming Languages*, ACM SIGPLAN Notices, Vol. 17, No. 12, Dec. 1982.

[Grundlage für diesen Teil]

J. **Bloch**, *Effective Java Programming Language Guide, 2nd Ed.*, Addison Wesley, 2008.

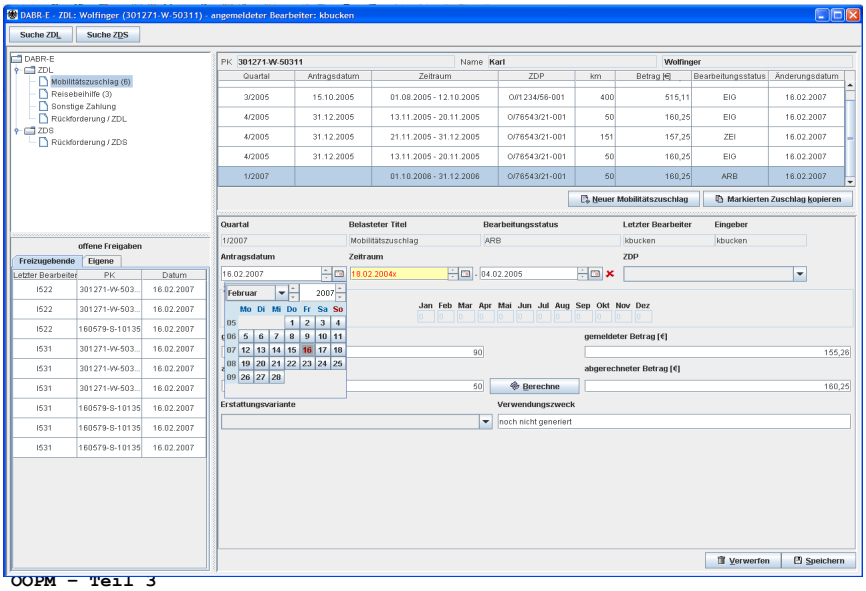
[Kenntnisreiche Darstellung der Fußangeln von Java; ein Muss für professionelle Java-Entwickler]

Motivation

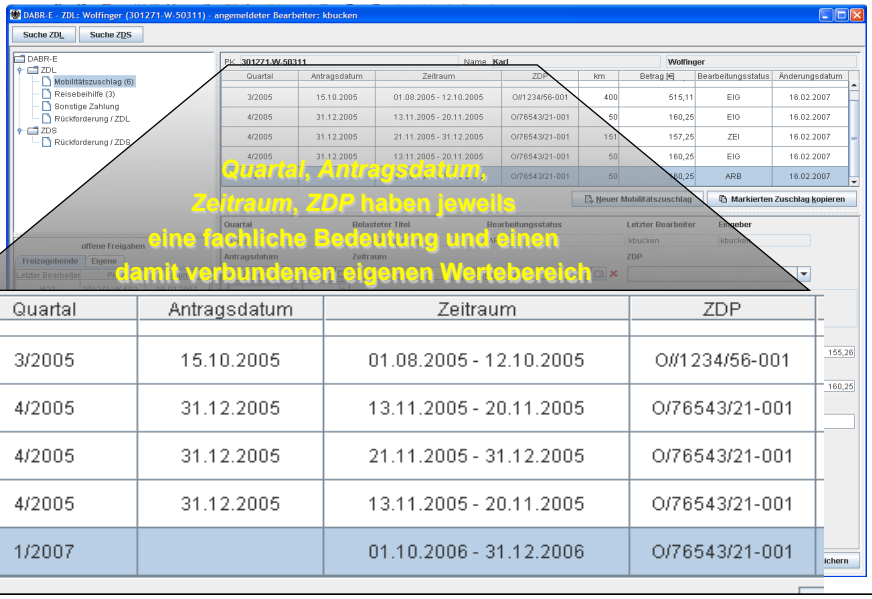
Programmieren ist Modellieren.

Wenn wir Software entwickeln, modellieren wir einen Ausschnitt der Welt.

Wir erinnern uns:
Menschen arbeiten mit Werkzeugen an Materialien



Bei genauer Betrachtung:
Materialien enthalten fachlich motivierte Werte



Quartal	Antragsdatum	Zeitraum	ZDP
3/2005	15.10.2005	01.08.2005 - 12.10.2005	O/I1234/56-001
4/2005	31.12.2005	13.11.2005 - 20.11.2005	O/76543/21-001
4/2005	31.12.2005	21.11.2005 - 31.12.2005	O/76543/21-001
4/2005	31.12.2005	13.11.2005 - 20.11.2005	O/76543/21-001
1/2007		01.10.2006 - 31.12.2006	O/76543/21-001

These aus Sicht des Werkzeug & Material-Ansatzes

In jedem Anwendungsbereich können wir fachlich motivierte Werte erkennen.

Zumindest fällt es schwer, die folgenden Phänomene als **Objekte** zu modellieren:

- einen Zeitpunkt
- einen Zeitraum
- einen Geldbetrag
- eine Kontonummer
- eine Sozialversicherungsnummer (SVN)

Zeitpunkte werden nicht erzeugt!

Wenn wir den Zeitraum für unsere Klausur „ändern“, dann wählen wir eigentlich nur einen anderen Zeitraum.

„100 Euro“ sind sogar in doppelter Hinsicht ein Wert.

Wenn sich bei einem Bankkunden die Kontonummer „ändert“, dann ändert sich lediglich die Zuordnung zwischen Kunde und Nummer – nicht die Nummer selbst.

Wenn ein neuer Mitbürger eine SVN bekommt, dann wird diese nicht frisch erzeugt, sondern es wird lediglich einer Person eine gültige SVN zugeordnet.

Mathematische Werte in der Programmierung

- Ganze Zahlen → Zweierkomplement (`int`)
- Rationale Zahlen → Gleitkommazahlen (`float`)
- Boolesche Wahrheitswerte (`boolean`)

- Sind das Objekte?
  - Wann haben wir je eine „4“ erzeugt? Können wir die „4“ verändern?
  - Wurde die Zahl Pi entdeckt oder erzeugt?
  - Wurde `true` vor `false` erzeugt? Ist das wichtig?

- Die primitiven Typen in Sprachen wie Java scheinen eindeutig eher Werte zu modellieren…

Wann verwenden wir Zahlen?

- Wir **verwenden Zahlen als eine Form von Werten**:
  - zur **Identifikation** (Bezeichnung) von modellierten Gegenständen (Bsp.: Kontonummer, Personalnummer),
  - zum **Abzählen** und **Ordnen** von Gegenständen außerhalb und im Rechner,
  - zur **Repräsentation** von messbaren **Größen**.
- Im Bereich **Naturwissenschaften und Technik**:
  - zur **numerischen Analyse**, d.h. zur Lösung mathematischer Probleme durch Operationen auf Zahlen.
- Alle Zahlen sind Werte – aber sind alle fachlichen Werte auch Zahlen?

Was unterscheidet fachliche Werte von Zahlen?

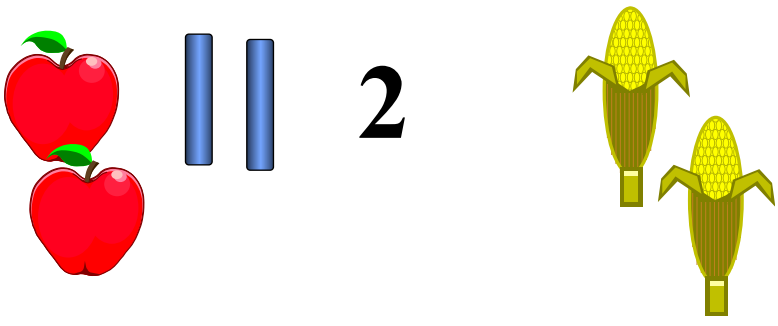
- **Fachwerte** gehören oft zu einem Anwendungsbereich und haben eine fachliche Bedeutung.
  - Beispiele: **Postleitzahl**, **Bankleitzahl**
- Fachliche Werte haben oft Operationen, die von den allgemeinen numerischen Operationen abweichen. Beispiele:
  - **Kontonummern** werden manchmal **addiert** (Prüfsumme für Überweisungen), aber nicht **subtrahiert**.
  - **Geldbeträge** können wir **addieren** u. **subtrahieren**, aber nicht miteinander **multiplizieren**.
- Nicht alle fachlichen Werte lassen sich geeignet durch Zahlen modellieren – wie beispielsweise die Grundfarben: **{ Rot, Grün, Blau }**.
- In **Programmen** sollten (programmiersprachliche) Werte bestimmte Eigenschaften besitzen, die sie klar von **Objekten** unterscheiden.

Vergleich: Wert versus Objekt

- Auf den folgenden Folien stellen wir die Begriffe **Wert** und **Objekt** einander gegenüber:
  - Was sind die zentralen Eigenschaften von Werten und Objekten?
  - Worin unterscheiden sich Objekte und Werte voneinander?
- Ziel ist ein klareres Verständnis dieser fundamentalen Begriffe.

Wert: Abstrakt

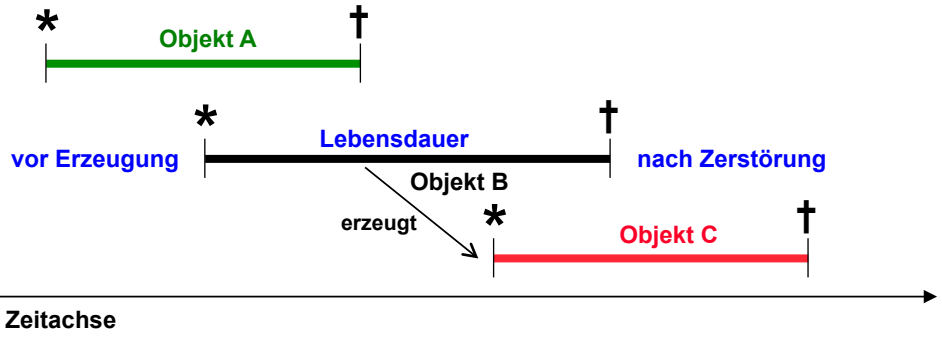
- Ein **Wert** ist **abstrakt**:
  - Werte sind immer immateriell.
  - Werte abstrahieren von **konkreten Kontexten**.
  - **Fachliche Werte** sind häufig Abstraktionen von Dingen, um diese zu identifizieren (Kontonummer, Postleitzahl).



© MacLennan

Objekt: Erzeugbar und zerstörbar

- Ein **Objekt** kann **erzeugt** und **zerstört** werden:
  - Da Objekte in der Zeit existieren, haben sie einen **zeitlichen Anfang** und ein **Ende**, also eine **Lebensdauer**.
  - Aus Sicht eines Objektes sind drei Zeitabschnitte relevant: Vor seiner Erzeugung, während seiner Lebenszeit und nach seiner Zerstörung.



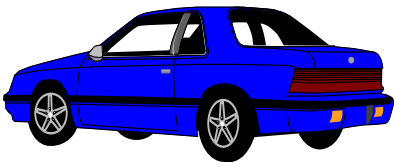
Wert: Zeitlos

- Ein **Wert** ist **zeitlos**:
  - Begriffe wie **Zeit** und **Dauer** sind nicht anwendbar.
  - Werte werden nicht **erzeugt** oder **zerstört**.
  - In Ausdrücken **entstehen** keine Werte und sie werden nicht **verbraucht**.

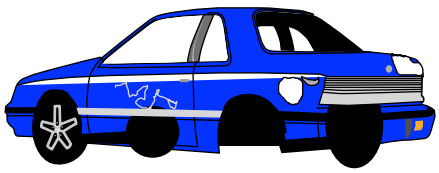
```
40 + 2 = 42
```

Objekt: Potenziell veränderbar

- Ein **Objekt** kann **veränderbar** sein, d.h.:
  - die **erkennbaren Merkmale** können sich ändern,
  - trotz **Veränderung** bleibt die **Identität** erhalten.



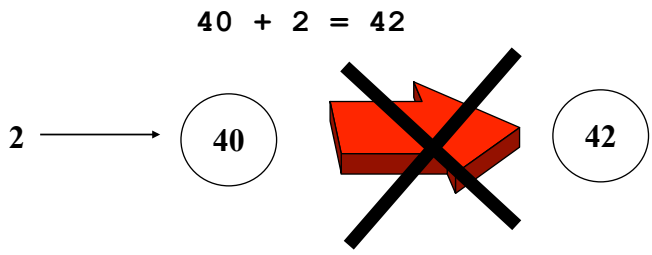
Mein Auto 1995



Mein Auto 2004

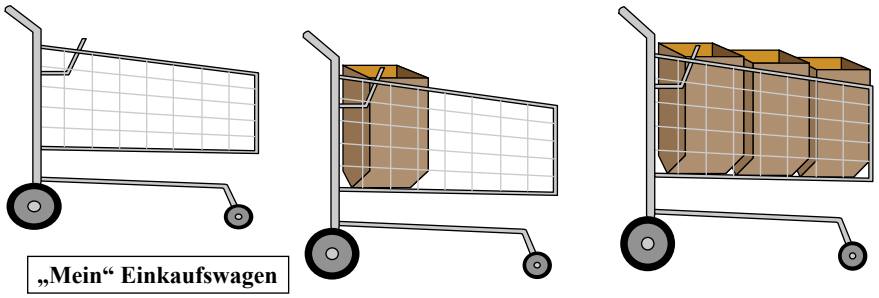
Wert: Unveränderlich

- Ein **Wert** ist **unveränderlich**:
  - Er kann **berechnet** und **auf andere Werte bezogen** werden, aber **nicht verändert** werden.
  - **Funktionen** können (applikativ) auf Werte angewandt werden, um **andere Werte** zu berechnen.



Objekt: Zustandsbehaftet

- Ein **Objekt** hat einen (inneren) **Zustand**:
  - **Veränderung** eines Objekts bedeutet die Veränderung seines **Zustandes**.
  - **Veränderungen** finden „in der **Zeit**“ statt.



„Mein“ Einkaufswagen

Es gibt auch **unveränderliche Objekte**: Diese erhalten ihren Zustand einmalig bei ihrer Erzeugung.

SE2 – OOPM – Teil 3 21

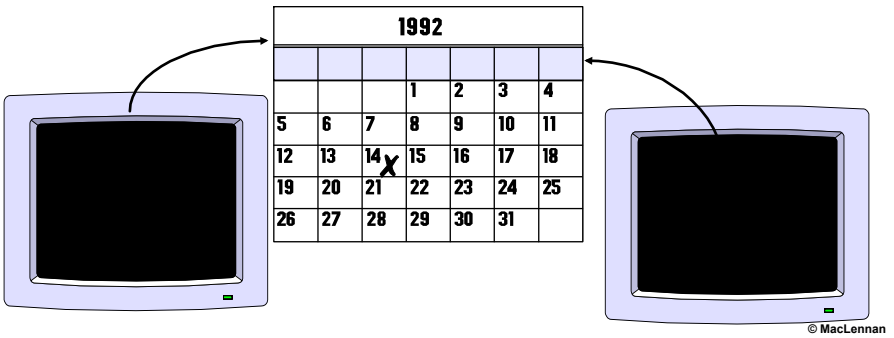
Wert: Zustandslos

- Ein Wert ist **zustandslos**:
  - Operationen berechnen Werte; sie **verändern** sie aber **nicht**.
  - Da der Zeitbegriff auf Werte nicht zutrifft, können sie sich auch nicht „mit der Zeit“ verändern.
  - Zustände lassen sich zwar mit Hilfe von Werten modellieren (Kontostand), aber ein Wert selbst hat keinen Zustand.

SE2 – OOPM – Teil 3 22

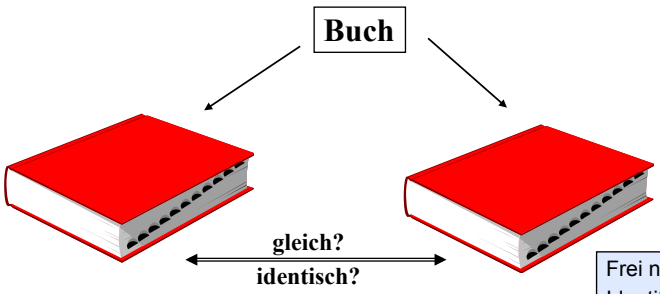
Objekt: Zur Kommunikation benutzbar

- Ein **Objekt** kann zur **Kommunikation** und **Kooperation** benutzt werden:
  - **Verändert** ein Benutzer den Zustand eines Objekts, ist dies für andere Benutzer erkennbar.
  - Dieser „**Seiteneffekt**" kann gewünscht oder problematisch sein.



Objekt: Unterschied zwischen Gleichheit und Identität

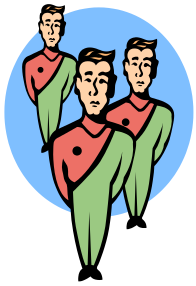
- Zwei **gleiche** Objekte sind nicht notwendig **identisch**:
  - **Gleichheit** bezieht sich auf die **erkennbaren Merkmale**,
  - **Identität** auf die (äußeren) **Zusammenhänge** eines Objekts (Position in Raum und Zeit).



Frei nach Frege:
Identität ist die Beziehung, in der ein Objekt mit sich selbst und mit keinem anderen steht.

Objekt: Potenziell kopierbar

- Begriffe wie „**Original**“ und „**Kopie**" sind bei Objekten fachlich oft sinnvoll.



- Von Werten gibt es **keine Kopien**.
- Der Begriff **Anzahl** ist auf einen Wert nicht anwendbar.

Bis hierher: Wert versus Objekt

- Ein **Wert** ist:
  - abstrakt
  - zeitlos
  - unveränderlich

- Ein **Objekt** ist:
  - zustandsbehaftet (potenziell veränderbar)
  - erzeugbar und zerstörbar (existent in Raum und Zeit)

Zwischenergebnis

- **Werte** und **Objekte** sind **Grundkonzepte** der Softwareentwicklung.
- Das Verständnis dieser Konzepte hilft beim **Entwurf** und bei der **Konstruktion** von Software.
- Wir sollten beim Modellieren eines Anwendungsbereichs die **Unterschiede** zwischen Werten und Objekten sehr gut kennen.
- Die **Programmiersprache** als wichtigstes Software-Werkzeug sollte uns dabei **unterstützen**, **sowohl Werte als auch Objekte** in unseren Programmen geeignet abzubilden.

- Frage: Wie gut machen die aktuell verwendeten **objektorientierten Sprachen** dies?

Objektorientierte Sprachen und Werte

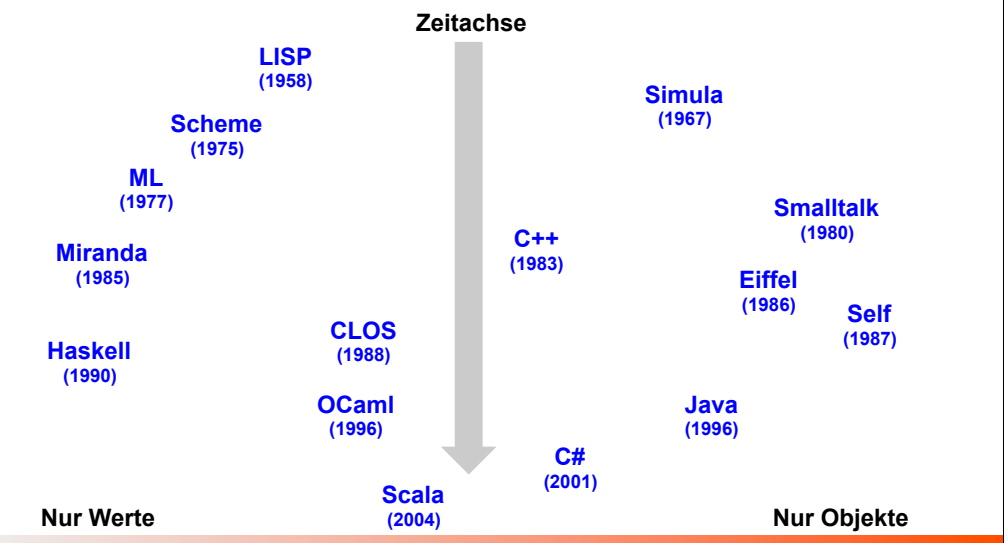
- Objektorientierte Sprachen erlauben mit dem Klassenkonstrukt die Definition beliebiger **Objekttypen**.
  - Da mit dem Schlüsselwort `class` ein neuer Typ konstruiert wird, sprechen wir auch von einem **Typkonstruktor**.
- Bei den **Werttypen** hingegen ist die Menge innerhalb der Sprache üblicherweise festgelegt.
  - **Beispiel Java**: `int`, `byte`, `short`, `long`, `float`, `double`, `boolean`, `char`
- Wenn in vielen Anwendungszusammenhängen fachlich motivierte Werttypen nützlich sind, warum bieten objektorientierte Sprachen dann so wenig Unterstützung für benutzerdefinierte Werte?
  - Vermutlich wegen des Strebens nach **Purismus**…

Puristisch: Alles nur Objekte

„Pure“ objektorientierte Programmiersprachen:

- **Smalltalk**
  - Maxime: **Everything is an object!** Klassen, Werte, Blöcke…
- **Eiffel**
  - Zumindest zur Laufzeit nur Objekte
- **Self**
  - Keine Klassen, nur noch Objekte
  - neue Objekte entstehen durch Klonen

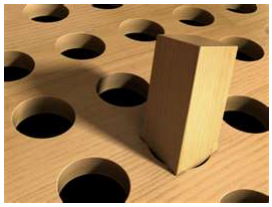
Werte und Objekte in Programmiersprachen: Spektrum



Alles nur Objekte...

- … was spricht dafür?
  - Einfaches Sprachmodell
  - Konsistente Semantik (insbesondere bei Variablen)
  - Einheitliche Generizität
  - …
- Dies sind primär Argumente für die Sprachdesigner, aber nicht unbedingt für Programmierer; auf keinen Fall jedoch für Modellierer.
-
- **Wenn wir gezwungen werden, alle relevanten Abstraktionen eines Anwendungsbereiches mit Objekten zu modellieren, dann verbiegen wir uns bei den Werten!**





Von Objekten zu Typen…

- In objektorientierten Programmiersprachen modellieren wir **Objekte**, indem wir **Klassen** definieren, von denen Exemplare erzeugt werden können.
- Jede Klasse definiert einen **Typ**, die Operationen der Klasse sind auch die **Operationen** des Typs.
- Die Exemplare einer Klasse bilden (extensional) die **Elementmenge** ihres **Objekttyps**.
- Wir nennen diese Menge explizit Elementmenge, da der sonst übliche Begriff „Wertemenge“ gerade bei der Diskussion über Werte und Objekte verwirrend sein kann. Sowohl Werte als auch Objekte bezeichnen wir im folgenden als **Elemente** ihres Typs.

Wikipedia: Die **Extension** eines **Begriffs** (z.B. „Mensch“) […] ist sein **Umfang**, das heißt die **Menge** aller Objekte („Erfüllungsgegenstände“), die unter diesen Begriff fallen.

…und von Werten zu Typen

- **Werte** wie die ganzen Zahlen oder die Wahrheitswerte werden in Programmiersprachen durch vordefinierte **Werttypen** (Java: `int`, `boolean`) modelliert.
- Die **Elementmenge** dieser Typen ist **unveränderlich**: Der Werttyp `int` verfügt über eine Wertemenge mit 2^{32} Elementen, während `boolean` zwei Elemente hat.
- Wir stellen verallgemeinernd fest: Wenn Werte konzeptuell nicht erzeugt und vernichtet werden, dann ist zwangsläufig die **Elementmenge bei Werttypen allgemein unveränderlich**.
- Wir sprechen deshalb bei Werttypen ungern von „Konstruktoren“, denn Werte werden ja nicht erzeugt; stattdessen nennen wir Operationen, die uns die Werte eines Werttyps liefern, **Selektoren**. Sie selektieren einen Wert aus der Menge der bestehenden Werte.

Zwischenstand: Wert, Objekt und der Typbegriff

- Ein **Wert** ist:
  - abstrakt
  - zeitlos
  - unveränderlich

- Ein **Objekt** ist:
  - zustandsbehaftet (potenziell veränderbar)
  - erzeugbar und zerstörbar (existent in Raum und Zeit)

in Programmiersprachen definiert über

- **Werttypen**:
  - haben eine unveränderliche Elementmenge
  - bieten Selektoren an

- **Objekttypen**:
  - haben eine dynamische Elementmenge
  - bieten Konstruktoren an

- **Gemeinsam**:
  - Typ = Elementmenge + Operationen
  - Variablen des Typs mit veränderbarer Belegung

Die Frage der Gleichheit

- Beim **Gleichheitsbegriff** bestehen in Programmiersprachen deutliche Unterschiede zwischen Werten und Objekten.
- Beispiel Java: Was bedeutet a == b ?
- Betrachte:

```
a = b;
if (a == b) { ... }
a = 42;
b = 42;
if (a == b) { ... }
```

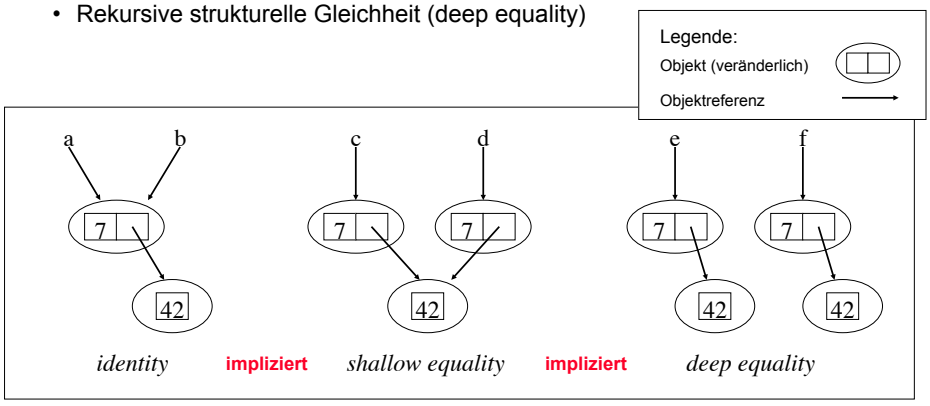
- Wir würden uns wünschen, dass a und b gleich sind!
- Was ist dann mit Strings?
- Betrachte:

```
name = "Elling";
if (name.toLowerCase() == "elling") {
  ...
}
```


Gleichheit für Objekte

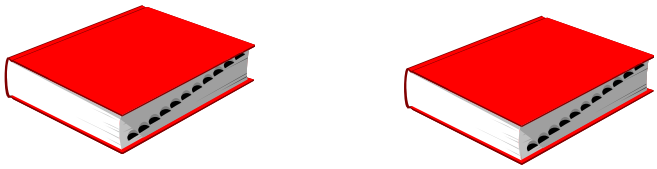
Technisch sind drei Formen unterscheidbar:

- Referenzgleichheit, Identität (identity)
- Einfache strukturelle Gleichheit (shallow equality)
- Rekursive strukturelle Gleichheit (deep equality)



Technische vs. semantische Gleichheit

- Alles einfach zu implementieren, aber nicht ausreichend.
- Betrachte:
  Bibliotheksbuch:
  Titel Autor ISBN RefNummer
  - kann als gleich angesehen werden, auch wenn die Referenznummer unterschiedlich ist...
- Gebraucht wird deshalb häufig auch ein Konzept von **semantischer Gleichheit**.



Aber es gibt doch benutzerdefinierte Gleichheit…

- … wie beispielsweise über das `equals` in Java. Damit können wir für jeden Objekttyp definieren, wann Objekte als gleich gelten sollen!
- Ist das die Lösung?
- Die Prüfung auf Gleichheit ist dann ein Aufruf einer Operation:
  - asymmetrisch
  - Vergleich mit `null`?
  - Welchen Typ hat der Parameter?
    - `Object` wie in Java? Das führt zu Downcasts…
    - `like Current` wie in Eiffel? Ist nicht statisch typsicher…

Gleichheit in Theorie und Praxis

Mathematisch formuliert:

- Reflexiv
  a = a (Identität impliziert Gleichheit)
- Symmetrisch
  a = b => b = a
- Transitiv
  a = b ^ b = c => a = c

Wir halten fest:

- Bei Objekten besteht ein **Unterschied** zwischen **Gleichheit** und **Identität**; bei Werten hingegen ist diese Unterscheidung unüblich.

Transitivität seit Java 1.5...

Auto-Boxing und -Unboxing:

```
Integer a = 128;
int b = 128;
Integer c = 128;
```

Es gilt:

```
a == b
b == c
```

Aber:

```
a != c
```

Fazit: Wert, Objekt und der Typbegriff

- Ein **Wert** ist:
  - abstrakt
  - zeitlos
  - unveränderlich

- Ein **Objekt** ist:
  - zustandsbehaftet (potenziell veränderbar)
  - erzeugbar und zerstörbar (existent in Raum und Zeit)

in Programmiersprachen definiert über

- **Werttypen**:
  - haben eine unveränderliche Elementmenge
  - bieten Selektoren an

- **Objekttypen**:
  - haben eine dynamische Elementmenge
  - bieten Konstruktoren an
  - unterscheiden Gleichheit und Identität

- **Gemeinsam**:
  - Typ = Elementmenge + Operationen
  - Variablen des Typs mit veränderbarer Belegung

Aufgepasst: Unveränderliche Objekte vs. Werte

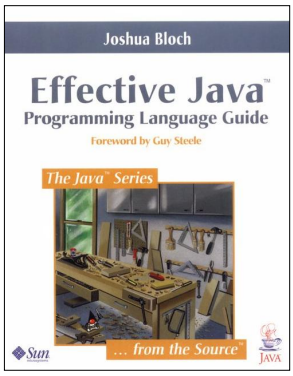
- In vielen Anwendungskontexten ist es sinnvoll, **unveränderliche Objekte** zu definieren.
- Beispiel: Bei einem Film in einer Filmdatenbank sind **Titel**, **Regisseur** und **Filmlänge** unveränderlich, ein Film kann somit durch eine Objektklasse **Film** definiert werden, deren Exemplare diese Eigenschaften bei der Erzeugung eines Filmobjektes übergeben bekommen und sie anschließend nur über lesende Operationen zur Verfügung stellen.
- Wenn ein **Objekt ausschließlich sondierende Operationen** anbietet – ist es dann nicht quasi ein Wert?
- Die Antwort lautet **Nein**: Denn für einen Wert müssen **alle konzeptionellen Eigenschaften** zutreffen. Ein Film ist sicher etwas Abstraktes und kann auch als etwas Unveränderliches angesehen werden; aber ein Film ist nicht zeitlos, denn es gibt einen Zeitpunkt, bis zu dem der Film nicht existierte und ab dem der Film dann (konzeptionell) für immer existiert.

In ferner Zukunft: benutzerdefinierte Werttypen

- Zukünftige objektorientierte Programmiersprachen sollten ein dediziertes **Konstrukt** anbieten, mit dem eigene **Werttypen** definiert werden können.
- Anforderungen an die **Werttypen**, die durch einen solchen Typkonstruktor definiert werden können, sind nach der vorangegangenen Diskussion:
  - Ihre **Wertemenge** sollte **fixiert** sein; da Werte nicht erzeugt und zerstört werden können, kann es beispielsweise keine Konstruktoren für Werte geben.
  - Ihre **Elemente** müssen **unveränderlich** sein (am besten garantiert durch den Compiler).
  - Ihre **Operationen** sollten **referentiell transparent** sein und **keine beobachtbaren Seiteneffekte** zeigen.
  - Sie sollten **ausschließlich auf der Basis anderer Werttypen definiert** werden (Werte können sich nicht auf Objekte beziehen).

Werttypen in Java mit Wertklassen

- Da Java keine explizite Unterstützung für benutzerdefinierte Werttypen bietet, müssen wir mit den vorhandenen „Bordmitteln“ auskommen.
- Wir müssen deshalb Werttypen **mit Objektklassen definieren** (im Folgenden **Wertklassen** genannt) und dabei darauf achten, dass die Werteigenschaften eingehalten werden.
- Es gibt etliche **Programmiermuster**, die uns dabei unterstützen.
- Prominentes Beispiel:
  - Das **Typesafe Enum Pattern** von Bloch.



Das Typesafe Enum Pattern

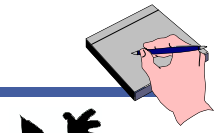
- Beschreibt, wie **Aufzählungstypen** typsicher in Java modelliert werden können.

  Grundmuster:
  - Biete ein `public static final` Feld für jede Konstante der Aufzählung.
  - Verstecke den Konstruktor.

```
public class Farbe {
    private final String _name;

    private Farbe(String name) { _name = name; }

    public String toString() { return _name; }

    public static final Farbe ROT = new Farbe("rot");
    public static final Farbe GRUEN = new Farbe("gruen");
    public static final Farbe BLAU = new Farbe("blau");
    ...
}
```

- Ist inzwischen in die Sprache eingeflossen (seit Java 1.5): Hinter den Kulissen der **Enums** in Java wird dieses Muster realisiert.
- Die Enums in Java sind allerdings keine „waschechten“ Werttypen, da an den Exemplaren eines Enums ein **veränderlicher Zustand** modelliert werden kann.
- Es entstehen **Wertobjekte** – fachlich motivierte Werte, die technisch durch Objekte repräsentiert werden müssen.

Sechs Richtlinien zu Wertklassen in Java



Konstruktion von Wertklassen

Stelle sicher,

1. dass Wertobjekte keinen veränderbaren Zustand haben;
2. dass Werte und Wertobjekte sich nicht auf Objekte beziehen;
3. dass im Quelltext einer Wertklasse keine (bestehenden) Objekte verändert werden.
4. Verberge die (technisch notwendige) Erzeugung von Wertobjekten.
5. Implementiere `equals` und dazu passend `hashCode`.

Benutzung von Wertklassen

6. Verwende immer `equals` statt `==` bei der Prüfung auf Gleichheit zweier Werte.

1. Unveränderlicher Zustand

- Die Wertobjekte von Wertklassen sollten unveränderlich sein.
- **Sprachunterstützung**
  - Mit dem Schlüsselwort `final` kann für **Zustandsfelder** festgelegt werden, dass sie unveränderlich sein sollen.
    - Einem Zustandsfeld, das als `final` gekennzeichnet ist, darf nur innerhalb des Konstruktors ein Wert zugewiesen werden.
  - Mit dem Schlüsselwort `final` kann für die gesamte **Wertklasse** festgelegt werden, dass es keine Subklassen geben darf.
    - Auf diese Weise wird verhindert, dass Unterklassen mit veränderlichem Zustand definiert werden können, deren Exemplare polymorph verwendet werden könnten.

2. Wertobjekte als Blätter im Objektbaum

- Die primitiven Werte in Java bilden bereits die Blätter im Objektbaum. Darüber hinaus sollten Wertobjekte keine Referenzen auf (echte) Objekte enthalten.
- **Sprachunterstützung**
  - keine

- **Selbst beachten**
  - In einer Wertklasse sollten die Typen aller Zustandsfelder nur Werttypen (primitive Typen oder Wertklassen) sein.
- String kann als eine Wertklasse angesehen werden.

3. Keine bestehenden Objekte verändern

- Wertobjekte sollten keine (echten) Objekte verändern.
- **Sprachunterstützung**
  - keine

- **Selbst beachten**
  - Im Quelltext einer Wertklasse sollten keine (bereits bestehenden) Objekte verändert werden.
  - In der Schnittstelle einer Wertklasse sollten deshalb keine Objekttypen als Parametertypen erscheinen.
  - Pragmatisch kann zugelassen werden, dass in einer Methode lokal Objekte erzeugt werden (`StringBuffer`, `Formatter`, etc.).

4. Erzeugung verbergen

- Die Kontrolle über die Erzeugung von Exemplaren einer Wertklasse sollte in der Wertklasse selbst liegen.

- **Sprachunterstützung**
  - Die Konstruktoren einer Klasse können als `private` deklariert werden. Klienten können dann nicht mehr direkt Exemplare der Klasse erzeugen.
  - Als Ersatz kann eine Klasse **Fabrikmethoden** anbieten: Klassenmethoden (mit `static` deklariert), die Exemplare der Wertklasse liefern. Diese Methoden dienen dann als **Selektoren**.

- Auf diese Weise ist es teilweise möglich, auf Regel 6 zu verzichten:
  - Wenn in der Wertklasse garantiert wird, dass für jeden Wert nur ein Wertobjekt erzeugt wird, dann können Klienten Referenzgleichheit verwenden (siehe die Enumerations in Java).

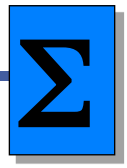
5. equals und hashCode implementieren

- Die Gleichheit für Wertobjekte sollte explizit definiert werden. Da verschiedene Wertobjekte denselben Wert repräsentieren können (keine Referenzgleichheit), muss die Gleichheit mit der Operation `equals` definiert werden.

- **Sprachunterstützung**
  - keine

- **Selbst beachten**
  - Der Vertrag des Typs `Object` für seine Operationen `equals` und `hashCode` muss eingehalten. U.a. gilt:
    - Zwei Wertobjekte, die als gleich gelten, müssen auch denselben Wert als Hash-Code liefern.

Zusammenfassung Wert und Objekt



- **Werte** und **Objekte** sind fundamental verschieden.
- **Zustandsbehaftete**, **vergängliche Gegenstände und Konzepte** lassen sich gut als **Objekte** modellieren.
- **Zeitlose und unveränderliche Größen** wie Zahlen und Zeiträume werden sinnvoll als **Werte** dargestellt.
- **Fachliche Werte** spielen in vielen Anwendungsbereichen eine große Rolle. Sie haben oft Operationen, die nicht den mathematischen Operationen auf Zahlen entsprechen.
- **Objektorientierte Programmiersprachen** stellen neben Objekten auch elementare Werte zur Verfügung. Sie bieten aber keine einfachen Möglichkeiten, **benutzerdefinierte fachliche Werte** einzuführen.