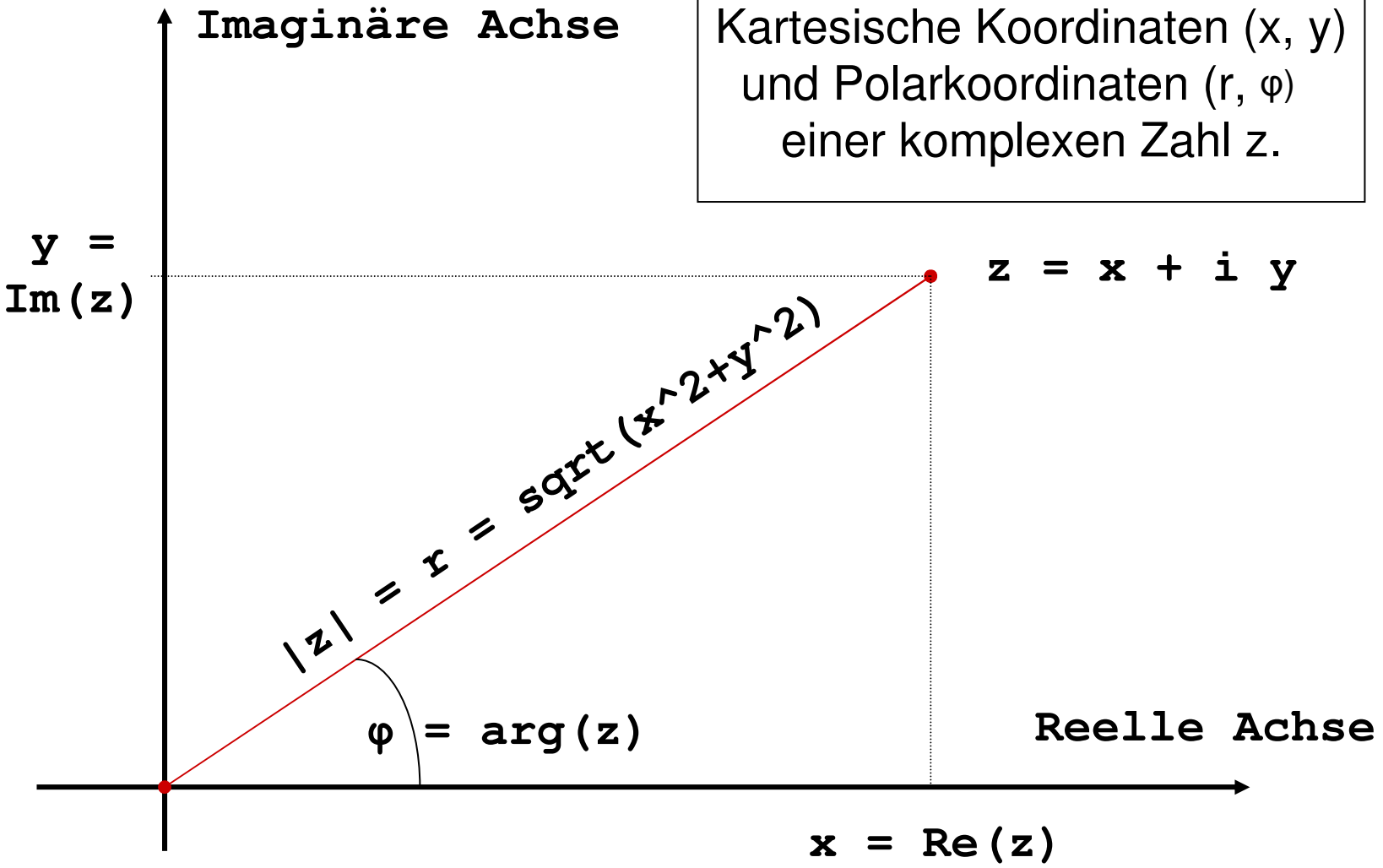
OOP

Objekt-orientiertes Programmieren
– nun genauer –

Erinnerung: Die Gauß'sche Zahlenebene



OOP in Ruby: Definieren einer Klasse

```
class Complexnum          # Unterklasse von "Object"
  def initialize(r=0.0, i=0.0)# Defaults: 0.0
    @re = r
    @im = i               # Attribute !
  end
end
```

```
a = Complexnum.new( 0.0, -1.0 )
b = Complexnum.new( 2.0 )
c = Complexnum.new
puts a, b, c              # Verwendet Object#to_s
```

```
#<Complexnum:0x2a67a08>
#<Complexnum:0x2a679a8>
#<Complexnum:0x2a67990>
```

OOP in Ruby: Definieren einer Klasse

- Bemerkungen
  - Der Konstruktor "new" wird von Ruby bereitgestellt. Er wird normalerweise nicht überladen.
  - "new" ruft "initialize" auf, welches Gelegenheit gibt, das soeben angelegte Objekt zu füllen.
  - Ruby kennt keinen Destruktor - nicht mehr benötigte (referenzierte) Objekte werden automatisch von der GC entfernt.

OOP in Ruby: Überladen einer Methode

```
# Erweitern einer bereits vorhandenen (!) Klasse:
#
class Complexnum  # Einfach nochmal "öffnen"...
  def to_s        # Standardmethode to_s überladen
    "(" + @re.to_s + ", " + @im.to_s + ")"
  end
end
```

```
# Erneut ausgeben:
puts a, b, c      # Verwendet nun Complexnum#to_s
```

```
(0.0, -1.0)
(2.0, 0.0)
(0.0, 0.0)
```

OOP in Ruby: Getter und Setter

```
class Complexnum
  def re       # Getter für @re
    @re
  end
  def im       # Getter für @im
    @im
  end
  def re=(v)   # Setter für @re
    @re = v
  end
  def im=(v)   # Setter für @im
    @im = v
  end
end
```

```
a.im           # -1.0
a.re = 3.0
puts a
```

```
(3.0, -1.0)
```

OOP in Ruby: Getter und Setter (Kurzform)

Entweder separat:

```
class Complexnum
  attr_reader :re, :im   # Legt die Getter an
  attr_writer :re, :im   # Legt die Setter an
end
```

oder gleich gemeinsam:

```
class Complexnum
  # Getter und Setter gleichzeitig anlegen:
  attr_accessor :re, :im
end
```

Kommentare:

- `:re` und `:im` sind Symbole

OOP in Ruby: Virtuelle Attribute

```
class Complexnum
  def abs      # Getter?
    # aus @re und @im berechnen ...
  end
  def arg      # Getter?
    # aus @re und @im berechnen ...
  end
  def abs=(v) # Setter?
    # @re und @im neu berechnen ...
  end
  def arg=(v) # Setter?
    # @re und @im neu berechnen ...
  end
end
```

- **OO-Trend "*Uniform access principle*"** (B. Meyer, 1997)**:**

  Die Grenzen von Methoden und Attributen verschwimmen! Neue Möglichkeiten für späteres "*refactoring*" ohne Konflikt mit Anwendern.

  Hier: Von außen ist nicht erkennbar, ob unsere Klasse intern mit Kartesischen oder Polar-Koordinaten arbeitet.

OOP in Ruby: Neue Methoden & Operatoren

```
# Erneut Erweitern der Klasse:
#
class Complexnum
  def +(z)              # Komplexe Addition
    Complexnum.new(@re + z.re, @im + z.im)
  end

  def absq              # Betragsquadrat
    @re * @re + @im * @im
  end
end
```

```
puts a, b, a+b, a.absq
```

```
(3.0, -1.0)
(2.0, 0.0)
(5.0, -1.0)
10.0
```

OOP in Ruby: Klassen-Methoden/-Attribute

```
# Erweitern der Klasse, initialize umdef.
#
class Complexnum
  @@zaehler = 0                # Klassenattribut!
  def initialize(r=0.0, i=0.0)
    @re = r; @im = i
    @@zaehler += 1
   end
  def Complexnum.counter # Klassenmethode!
    @@zaehler
   end
end
```

```
arr = [ Complexnum.new, Complexnum.new(1.0),
  Complexnum.new(1.0, 2.5), Complexnum.new ]
puts Complexnum.counter
```

```
4
```

OOP in Ruby: Vererbung

- **Keine** Mehrfach-Vererbung!
  - Ruby betrachtet M. als problematisch / zu vermeiden.
- Statt dessen: "Mixins"
  - Matz: "*Single inheritance with implementation sharing*"

- 2 Varianten der Einfach-Vererbung:

  1) Normale Vererbung:

```
class MySubclass < MyParentClass
  ...
end
```

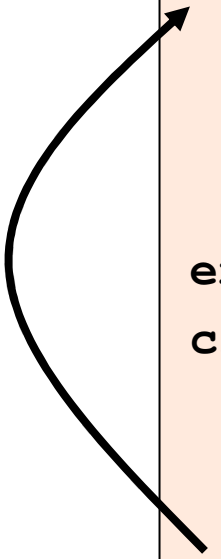
  Default:

```
class MyClass < Object
  ...
end
```

Optional!

OOP in Ruby: Vererbung

Verwendung von Methoden der Basisklasse: **super**



```ruby
class Person
  def initialize(name, geb_datum)
    @name, @geb_datum = name, geb_datum
  end
  # Weitere Methoden ...
end
class Student < Person
  def initialize(name, geb_datum, matr_nr, st_gang)
    @matr_nr, @st_gang = matr_nr, st_gang
    super(name, geb_datum) # initialize() von "Person"
  end
end
```

```ruby
a = Person.new("John Doe", "1950-01-01")
b = Student.new("Irgend Jemand", "1988-12-01",
                123456, "Allgemeine Informatik (BA)")
```

OOP in Ruby: Vererbung

- Normale Methodensuche
  - Ruby sucht zunächst in der aktuellen Klasse nach dem passenden Methodennamen
  - Wird dort nichts gefunden, setzt Ruby die Suche in der nächsthöheren Basisklasse fort, usw.
  - Exception "NoMethodError", falls Suche erfolglos.
- Die Wirkung von "super"
  - "super" (verwendet wie ein Methodenname) bewirkt, dass mit der Suche nach dem aktuellen Methodennamen in der direkten Basisklasse begonnen wird.

- **Erinnerung:** Abstraktionsmittel "*Ist-ein*-Beziehung"
  - Bsp.: Ein Cabriolet *ist ein* Auto. Ein Auto *ist ein* Fahrzeug.

(nach N. Josuttis, aus: B. Oesterreich, Objektorientierte Software-Entwickung, Oldenbourg, 2001)

Konsequenz für die OOP: **Objekte einer abgeleiteten Klasse sollten stets auch Exemplare aller ihrer Basisklassen sein!**

OOP in Ruby: Vererbung

- **Achtung - Designfehler!**

```
class Quadrat
  def initialize(a)
    @a = a
  end
  def flaeche
    @a * @a
  end
end
class Rechteck < Quadrat
  def initialize (a, b)
    @b = b
    super(a)
  end
  def flaeche
    @a * @b
  end
end
```

```
r = Rechteck.new(3, 4)
q = Quadrat(5)
r.flaeche     # 12 (ok)
q.flaeche     # 25 (ok)
# Funktioniert zwar ...

r.is_a? Quadrat  # true
q.is_a? Rechteck # false
# Offenbar unsinnig!

# Flächenberechnung
# ferner redundant
# implementiert!
```

OOP in Ruby: Vererbung

- **Korrekte Version:**

```
class Rechteck
  def initialize (a, b)
    @a, @b = a, b
  end
  def flaeche
    @a * @b
  end
end

class Quadrat < Rechteck
  def initialize(a)
    super(a, a)
  end
end
```

```
r = Rechteck.new(3, 4)
q = Quadrat(5)
r.flaeche     # 12 (ok)
q.flaeche     # 25 (ok)

r.is_a? Quadrat # false
q.is_a? Rechteck # true
# Viel besser!

# Methode "flaeche"
# komplett geerbt.
```

OOP in Ruby: Vererbung, Singletons

- **Singleton (Einzelstück)**
  - Ein Entwurfsmuster der Sorte „Erzeugungsmuster“ – also eine systematische Methode zur Erzeugung bestimmter Klassen.
  - Nützlich, wenn es von einer Klasse nur ein Exemplar geben darf

- Beispiele
  - Modellierung einer nur einmal vorhandenen Hardware, etwa von Maus oder Tastatur
  - Verwaltung einer systemweit eindeutigen Log-Datei

- In Ruby:
  - Auch zur Modellierung von Unikaten oder Ausnahmen unter ansonsten gleichartigen Exemplaren einer Klasse geeignet. Dazu werden Methoden eines einzelnen Objekts gezielt erzeugt oder überladen!

OOP in Ruby: Vererbung, Singletons

Singleton-Klassen:

```
a = "hallo"; b = a.dup # b ist echte Kopie von a
class <<a
  def to_s            # Überladen von to_s nur für a
    "Der Wert ist '#{self}'"
  end
  def zweifach        # Neue Methode, nur für a
    self + self
  end
end
```

```
a.to_s                # "Der Wert ist 'hallo'"
a.zweifach            # "hallohallo"
b.to_s                # "hallo"
b.zweifach            # NoMethodError!
```

OOP in Ruby: Vererbung, Singletons

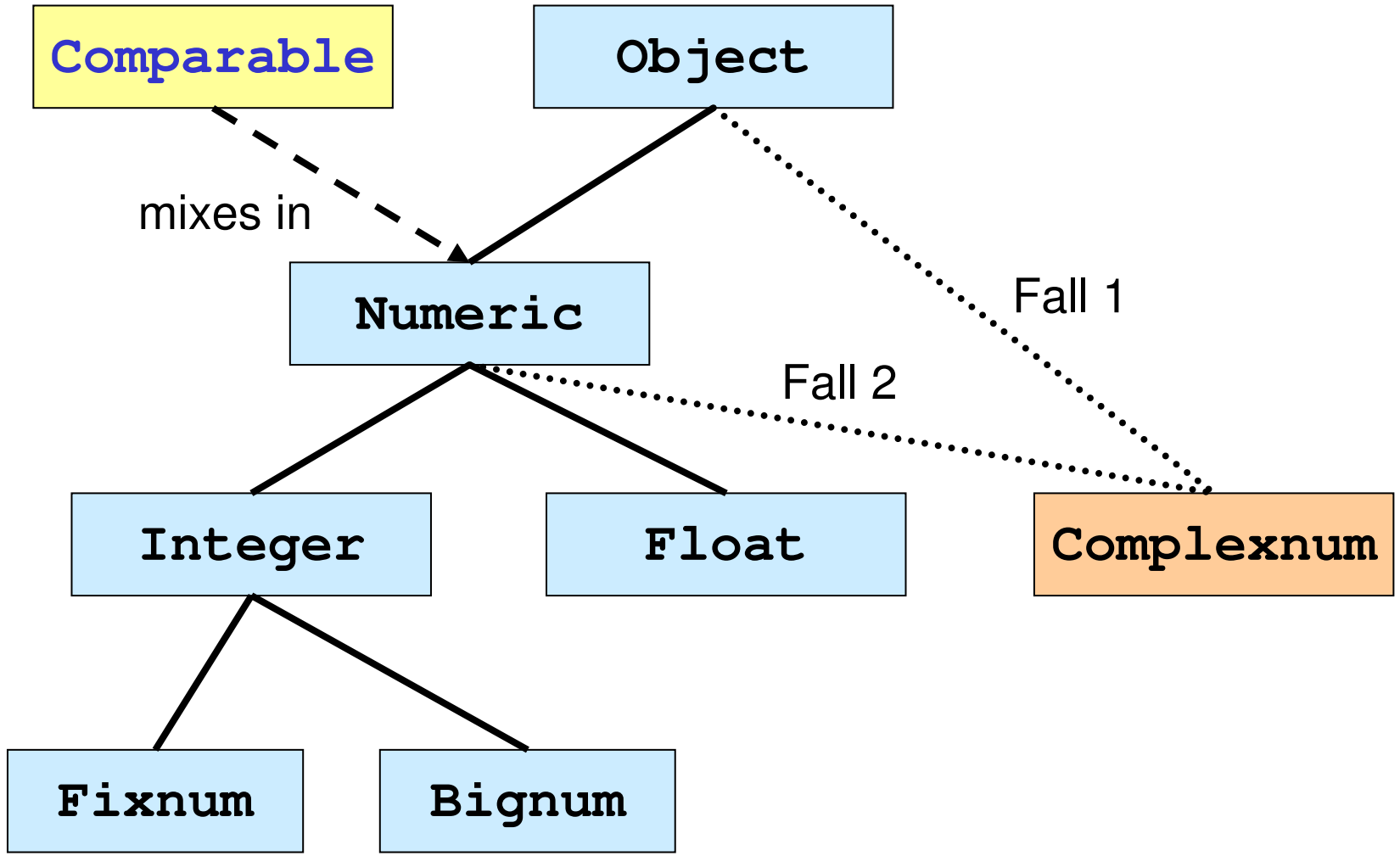
Singleton-Klassen, alternative Notation:

```
a = "hallo"
b = a.dup

def a.to_s          # Überladen von to_s nur für a
    "Der Wert ist '#{self}'"
end
def a.zweifach      # Neue Methode, nur für a
    self + self
end
```

```
a.to_s              # "Der Wert ist 'hallo'"
a.zweifach          # "hallohallo"
b.to_s              # "hallo"
b.zweifach          # NoMethodError!
```

OOP in Ruby: Vererbung, Klassenhierarchie



OOP in Ruby: Vererbung, "Selbstauskunft"

a = Complexnum.new(1.0, 2.0)			
Vergleichs-objekt	a.instance_of?	a.kind_of?, a.is_a? (1)	a.kind_of?, a.is_a? (2)
Numeric	false	false	true
Integer	false	false	false
Fixnum	false	false	false
Complexnum	true	true	true
Comparable	false	false	true

OOP in Ruby: Mixins, Bsp. "Comparable"

"*Implementation sharing*" in Aktion:

```
class Complexnum
  def <=>(z)          # Nur <=> implementieren
    if z.is_a? Complexnum # Konzept s.u.
       self.absq <=> z.absq
    else
       self.absq <=> z*z
    end
  end
end
```

```
puts a < b          # a, b: Complexnum-Objekte
false               # Operator '<' "geerbt"
```

– Implementieren von <=> erschließt automatisch:

```
<, >, <=, >=, ==, between?
```

OOP in Ruby: Mixins, Bsp. "Comparable"

- Das Beispiel ist mathematisch *nicht* sinnvoll
  - Komplexe Zahlen besitzen keine Ordnungsrelation!
  - Allerdings lassen sich ihre Beträge ordnen…

- Es zeigt aber die generelle Wirkung:
  - Implementieren weniger zentraler Methoden, plus
  - "Mixing in" eines Moduls wie "Comparable", das Methoden enthält, die nur auf diesen Methoden basieren

  ➔ Diverse Methoden werden auch für die neue Klasse quasi "geerbt", fast wie bei Mehrfachvererbung, Code-Redundanz wird vermieden.

OOP in Ruby: Mixins, Bsp. "Enumerable"

- Weiteres wichtiges Beispiel: `Enumerable`.

Benötigt:

`each`
`<=>`

Stellt bereit:

a) nur **`each`**:

`collect / map,`
`each_with_index,`
`find / detect,`
`find_all / select,`
`grep,`
`include? / member?,`
`reject,`
`to_a / entries`

b) plus **`<=>`**:

`max, min, sort`

OOP in Ruby: Mixins, Bsp. "Enumerable"

- Bemerkungen
  - "Enumerable" wird später vertieft (Vorstellung der Methoden), ebenso das Modulkonzept.
  - Die Selbstauskunfts-Methoden

    `kind_of?` bzw. `is_a?` (alias)

    akzeptieren auch "mixed in" Modulnamen, so als wären sie Elternklassen im Fall eines Mehrfachvererbungs-Modells:

```
class Complexnum < Numeric
...
end
a = Complexnum.new(1.0, 2.0)
a.is_a? Numeric              # true
a.kind_of? Comparable        # true !
```

Mehr zu OO in Ruby

Das Nachrichtenaustauschprinzip

Schutz von Methoden

Das Nachrichtenaustauschprinzip

- Das Nachrichtenaustauschprinzip:

> Objekte sind eigenständige Einheiten, deren Zusammenarbeit und Interaktionen mit Hilfe von Nachrichten bewerkstelligt wird, die sich die Objekte untereinander zusenden.
>
> B. Oesterreich, Objektorientierte Softwareentwicklung

- Übliche Umsetzung:
  - Nachricht = Methodenaufruf
- Beispiel in Ruby:

```
# Teil einer Methode "fahren" der Klasse "Autofahrer":
a = Car.new
a.start_engine # Autofahrer (self) sendet Nachricht an a
a.set_blinker("left") # weitere Nachricht, mit Param.
```

Das Nachrichtenaustauschprinzip

- Direkte Umsetzung des N.-Prinzips in Ruby:

```
# "Autofahrer"-Beispiel, einmal anders:
a = Car.new
# <self> sendet Nachricht "start_engine" an a
a.send( :start_engine ) # Per Symbol
a.send( "set_blinker", "left") # auch per Name!
```

- Auch Methoden selbst können in Ruby Objekte sein:

```
a = Car.new
start = a.method( :start_engine )
blink = a.method( "set_blinker" )
# <self> sendet Nachrichten:
start.call
blink.call( "left" )
```

  - Einzelheiten: Siehe Standardklasse "Method"

Schutz von Methoden

- **Zentrale Frage beim Thema Methodenschutz:**
  - **Wer darf welche Nachrichten an welche Objekte senden?**
- **Implementierung**
  - Methoden schützen & kapseln mit **Schlüsselwörtern**

    `private, protected, public`
  - Übernahme von Vorgehensweisen aus anderen OO-Sprachen
  - Zugriffsverletzungen entstehen ggf. nur zur Laufzeit!
- ***public***
  - Methoden sind standardmäßig "*public*", können also von jedem angewendet werden. Ausnahme: `initialize`
- ***protected***
  - "*protected*" Methoden einer Klasse C können nur von Objekten angewendet werden, die beide von C abstammen
- ***private***
  - "*private*" Methoden (auch von Elternklassen) können nur im funktionalen Stil vom Objekt auf sich selbst angewendet werden.

Schutz von Methoden

- **Anwendung**
  - Erste Form:
    Alle Methoden unterhalb eines der Schlüsselwörter unterliegen der jeweiligen Eingruppierung:

```
class ProtectionDemo

  def pub_meth1
    # ...
  end

protected

  def prot_meth
    # ...
  end

private

  priv_meth
    # ...
  end

public

  def pub_meth2(x)
    # ...
  end
end # class ProtectionDemo
```

- Bemerkungen:
  - pub_meth1 ist "public" per Default.
  - Die anderen Methoden unterliegen der direkten Regelung ihrer Eingruppierung.

Schutz von Methoden

- **Anwendung**
  - Zweite Form:
    Den Schlüsselwörtern folgen Listen der Symbole zu den jeweiligen Methoden:

```
class ProtectionDemo
  def pub_meth1
    # ...
  end

  def pub_meth2(x)
    # ...
  end

  def prot_meth
    # ...
  end

  def priv_meth
    # ...
  end

  public   :pub_meth1,
           :pub_meth2
  protected :prot_meth
  private  :priv_meth
end # class ProtectionDemo
```

- Bemerkungen:
  - Die Methode "initialize" einer jeden Klasse ist automatisch "private".

Schutz von Methoden

- **Beispiel, Vorbereitungen**

```
class C
  def … # wie Prot.Demo
  public   :pub_meth1,
    :pub_meth2
  protected :prot_meth
  private :priv_meth
end

class D < C
  def invoke(e)
    e.prot_meth
  end
end

class E < C
end

c,d,e = C.new,D.new,E.new
```

- **Beispiel, Tests**

```
class F < C
  def other_meth
    priv_meth # ok!
    self.priv_meth #FEHLER
  end
end

c.priv_meth # FEHLER

d.invoke(e) # ok

# Bricht ab:
F.new.other_meth
```

  - Ausführlicher in Demo: **prot_priv.rb**

Schutz von Methoden

- Beispiel "Kontoführung":

```
class Konten          # Vorgeschichte ausgelassen.
  private
       def belastung( konto, betrag )
         konto.wert -= betrag
       end
       def gutschrift( konto, betrag )
         konto.wert += betrag
       end
# Nur Methoden veröffentlichen, die vollständige
# Buchungen darstellen (Transaktionsschutz):
  public
       def spare( betrag )
         belastung( @gehaltskonto, betrag )
         gutschrift( @sparkonto, betrag )
       end
end
```

Abstrakte Klassen in Ruby?

- Abstrakte Klassen:
  - Klassen, die gemeinsame Methoden und Attribute für ihre Unterklassen enthalten, die aber selbst keine Exemplarbildung zulassen.
  - Ruby-Beispiel: **Integer**

    Diese Klasse besitzt keine (zugängliche) "new"-Methode:

```
# Normale Klassen:
String.methods.grep /new/  # ["new"]
Array.methods.grep /new/   # ["new"]

# "Abstrakte" Klasse:
Integer.methods.grep /new/ # []

# Andererseits …
Numeric.methods.grep /new/ # ["new"]
```

Sortieren

…per Mixin und dem Modul Enumerable

Effizienzgedanken

Sortieren

- Grundgedanke
  - Sortieren einer Liste erfordert eine Ordnungsrelation (<=) auf der Menge der Listenelemente
  - Sortieralgorithmen basieren auf paarweisen Vergleichen zwischen Listenelementen
- Konsequenz
  - Beliebige Listen lassen sich also mit effizienten Algorithmen sortieren, wenn sich zwei beliebige Listenelemente miteinander vergleichen lassen
- Ruby
  - Zu diesem Vergleich dient die Operatormethode `<=>`
  - Methode `sort` im Modul „Enumerable“ implementiert einen effizienten Sortieralgorithmus auf Basis von `<=>`
  - Beliebige Sortierwünsche lassen sich an `sort` per Codeblock übergeben. Dieser ermittelt den Vergleichswert zweier Listenelemente (-1, 0, 1, analog zu `<=>`)

Sortieren

- Beispiele, direkte Sortierung
  - Funktioniert, weil Klasse „String“ die Methode „<=>“ kennt, weil die Klasse „Array“ die Methode „each“ implementiert sowie das Mixin „Enumerable“ nutzt, welches „sort“ beisteuert:

```
%w/bc ab efg c/.sort          # ["ab", "bc", "c", "efg"]
```

```
[3,2,6,7,1].sort    # [1,2,3,6,7]
```
(auch Fixnum kennt **<=>**)

  - Sortierreihenfolge beeinflussen

```
[3,2,6,7,1].sort {|a,b| a <=> b}  # [1,2,3,6,7], s.o.
[3,2,6,7,1].sort {|a,b| b <=> a}  # [7,6,3,2,1]
```

```
# Zahlen in Hex-Darstellung numerisch sortieren:
%w/bc ab ef 8/.sort {|a,b| a.hex <=> b.hex}
```

```
# „Sortiert“ alles (wenn auch selten sinnvoll):
[1, 2.0, "3", {}].sort {|a,b| a.object_id <=> b.object_id}
```

Sortieren

- Effizienzgedanken
  - Manchmal ist die Ermittlung des Vergleichskriteriums „teuer“. Statt O(n log(n)) oder gar $O(n^2)$ Vergleiche auszuführen, sollte man dann die zu vergleichenden Ausdrücke mit Aufwand n berechnen und zwischenspeichern.
  - Methode „sort_by“ aus „Enumerable“ übernimmt die Detailarbeit:

```
customers.sort_by{|cust| cust.total_sales_in(2007)}
```

  - Auch hilfreich für mehrstufiges Sortieren (dank der Vergleichbarkeits-regeln von Arrays):

```
personen.sort_by do |p|
  [p.geburtsjahr, p.nachname, p.vorname]
end
# Sortiert zunächst nach Geb.-Jahr, dann nach
  Nachname, schließlich nach Vorname
```

  - VORSICHT: In einfachen Fällen ist **sort_by** viel „teurer“ als **sort**