Informatik I: Einführung in die Programmierung

17. Objekt-orientierte Programmierung: Einstieg

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel

6. Dezember 2016

UNI FREIBURG

Motivation

Was ist das?

UNI FREIBURG

- OOP ist ein Programmierparadigma (Programmierstil) – es gibt noch weitere.

Was ist das?

UNI FREIBURG

- OOP ist ein Programmierparadigma (Programmierstil) – es gibt noch weitere.
- Es ist die Art und Weise an ein Problem heranzugehen, es zu modellieren und somit auch zu programmieren.

Was ist das?

UNI FREIBURG

- OOP ist ein Programmierparadigma (Programmierstil) – es gibt noch weitere.
- Es ist die Art und Weise an ein Problem heranzugehen, es zu modellieren und somit auch zu programmieren.
- Bisher: Prozedurale Programmierung

Was ist das?

UNI FREIBURG

- OOP ist ein Programmierparadigma (Programmierstil) – es gibt noch weitere.
- Es ist die Art und Weise an ein Problem heranzugehen, es zu modellieren und somit auch zu programmieren.
- Bisher: Prozedurale Programmierung
  - Zerlegung in Variablen, Datenstrukturen und Funktionen

Was ist das?

UNI FREIBURG

- OOP ist ein Programmierparadigma (Programmierstil) – es gibt noch weitere.
- Es ist die Art und Weise an ein Problem heranzugehen, es zu modellieren und somit auch zu programmieren.
- Bisher: Prozedurale Programmierung
  - Zerlegung in Variablen, Datenstrukturen und Funktionen
  - Funktionen operieren direkt auf Datenstrukturen

Was ist das?

UNI FREIBURG

- OOP ist ein Programmierparadigma (Programmierstil) – es gibt noch weitere.
- Es ist die Art und Weise an ein Problem heranzugehen, es zu modellieren und somit auch zu programmieren.
- Bisher: Prozedurale Programmierung
  - Zerlegung in Variablen, Datenstrukturen und Funktionen
  - Funktionen operieren direkt auf Datenstrukturen
- Objektorientierung: Beschreibung eines Systems anhand des Zusammenspiels kooperierender Objekte

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!
- Sie können von uns als solche wahrgenommen werden.

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!
- Sie können von uns als solche wahrgenommen werden.
- Objekte haben

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!
- Sie können von uns als solche wahrgenommen werden.
- Objekte haben
  - in der realen Welt: Zustand und Verhalten

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!
- Sie können von uns als solche wahrgenommen werden.
- Objekte haben
  - in der realen Welt: Zustand und Verhalten
  - in OOP modelliert durch: Attributwerte bzw. Methoden

Objekte in OOP

UNI FREIBURG

- Der Zustand eines realen Objekts wird mit Hilfe von Attributwerten repräsentiert. Beispiel: Der *Kontostand* eines Kontos wird im Attribut `guthaben` als Zahl gespeichert.

Objekte in OOP

UNI FREIBURG

- Der Zustand eines realen Objekts wird mit Hilfe von Attributwerten repräsentiert. Beispiel: Der *Kontostand* eines Kontos wird im Attribut `guthaben` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert. Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `abheben` den Betrag, der unter dem Attribut `guthaben` gespeichert ist.

Objekte in OOP

UNI FREIBURG

- Der Zustand eines realen Objekts wird mit Hilfe von Attributwerten repräsentiert. Beispiel: Der *Kontostand* eines Kontos wird im Attribut `guthaben` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert. Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `abheben` den Betrag, der unter dem Attribut `guthaben` gespeichert ist.
- Methoden sind die Schnittstellen zur Interaktion zwischen Objekten.

Objekte in OOP

UNI FREIBURG

- Der Zustand eines realen Objekts wird mit Hilfe von Attributwerten repräsentiert. Beispiel: Der *Kontostand* eines Kontos wird im Attribut `guthaben` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert. Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `abheben` den Betrag, der unter dem Attribut `guthaben` gespeichert ist.
- Methoden sind die Schnittstellen zur Interaktion zwischen Objekten.
- Normalerweise wird der interne Zustand versteckt (Datenkapselung).

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
  - macht alleine praktisch gar nichts.

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
  - macht alleine praktisch gar nichts.

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
  - macht alleine praktisch gar nichts.

- Ein Objekt

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
  - macht alleine praktisch gar nichts.

- Ein Objekt
  - wird dem „Bauplan“ entsprechend erzeugt

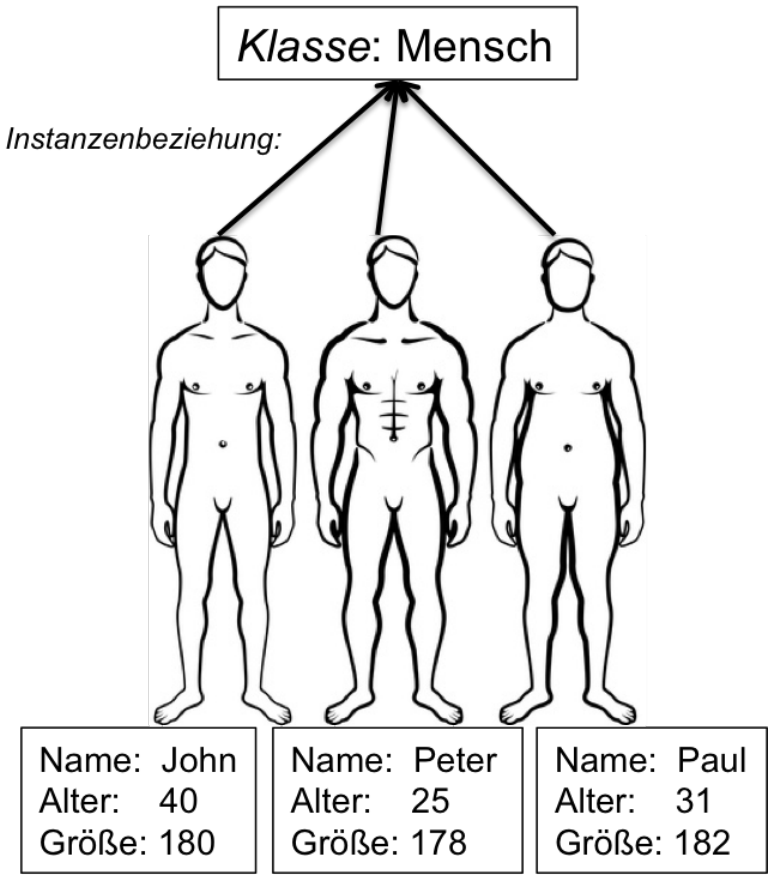
Klassen und Objekte (1)

UNI FREIBURG

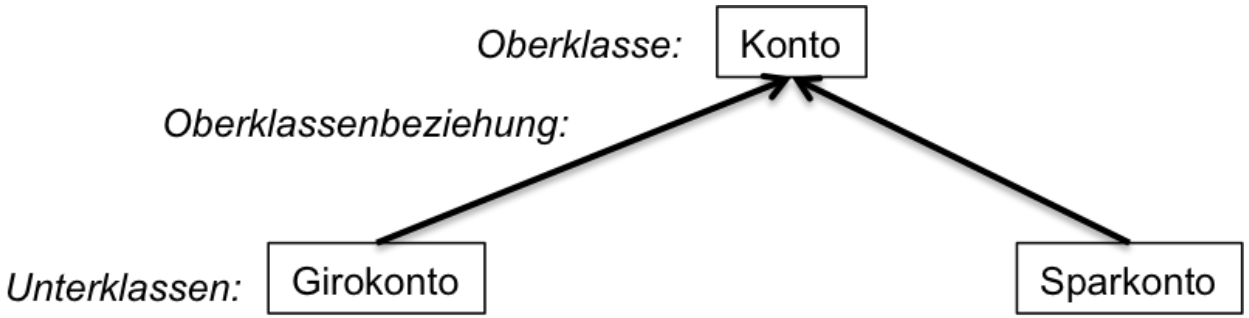
- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
  - macht alleine praktisch gar nichts.

- Ein Objekt
  - wird dem „Bauplan“ entsprechend erzeugt
  - ist dann ein Element/eine Instanz der Klasse

Klassen und Objekte (2)

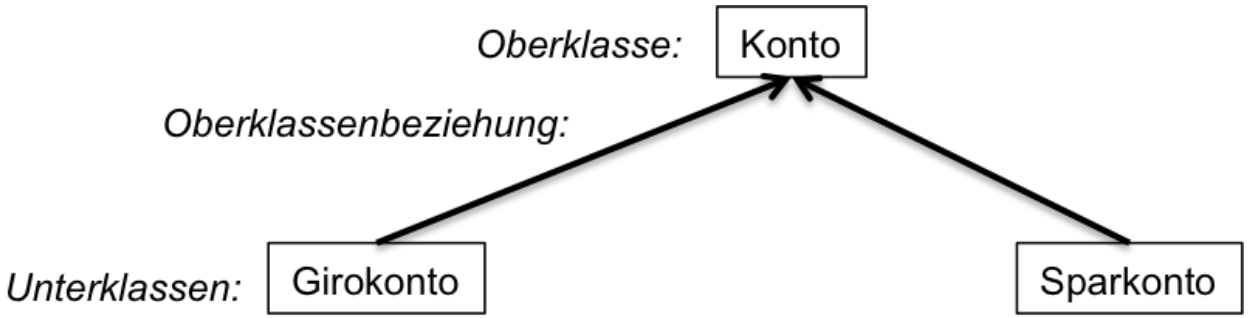


Klassenhierarchien



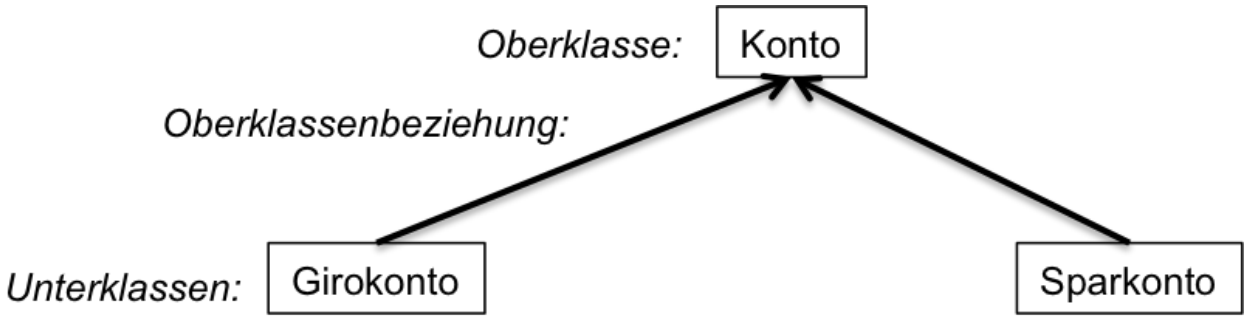
- Verschiedene Arten von Klassen können oft in einer Generalisierungshierarchie angeordnet werden:

Klassenhierarchien



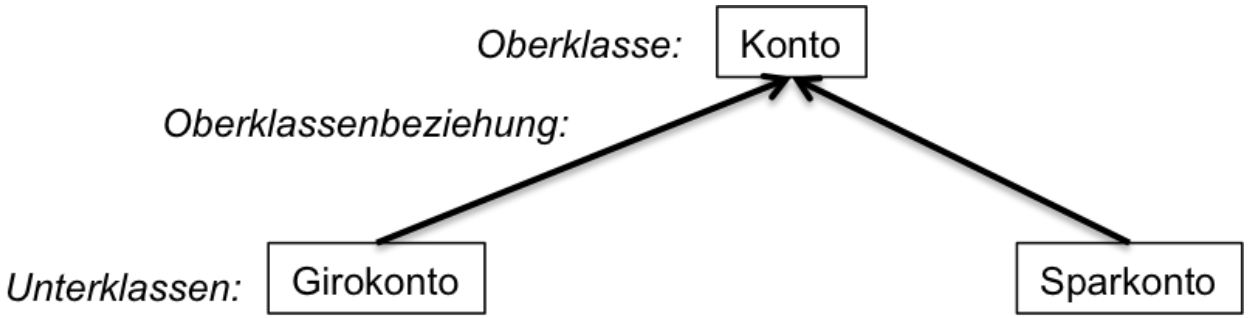
- Verschiedene Arten von Klassen können oft in einer Generalisierungshierarchie angeordnet werden:
- Man spricht von:

Klassenhierarchien



- Verschiedene Arten von Klassen können oft in einer Generalisierungshierarchie angeordnet werden:
- Man spricht von:
  - Superklasse, Oberklasse, Elternklasse und Basisklasse (für die obere Klasse)

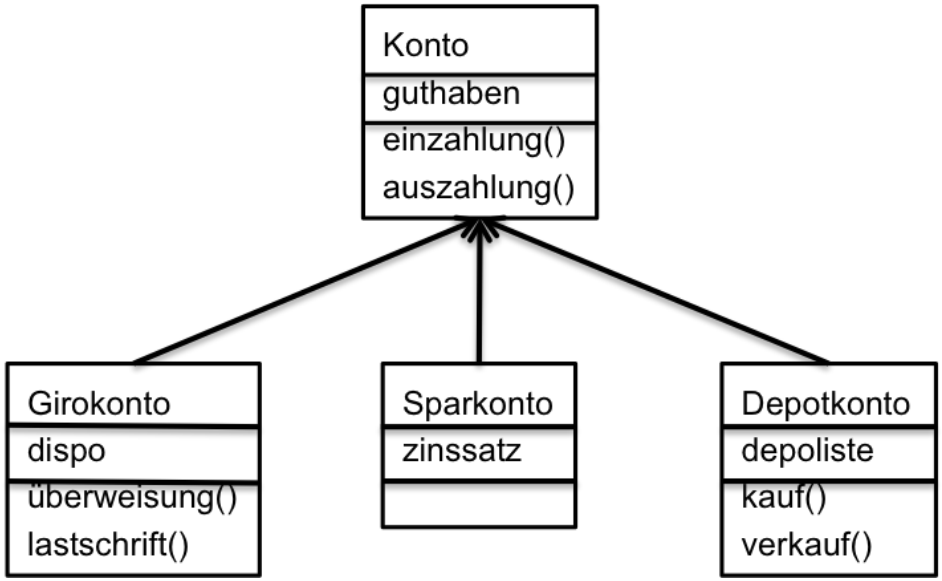
Klassenhierarchien



- Verschiedene Arten von Klassen können oft in einer Generalisierungshierarchie angeordnet werden:
- Man spricht von:
  - Superklasse, Oberklasse, Elternklasse und Basisklasse (für die obere Klasse)
  - Subklasse, Unterklasse, Kindklasse bzw. abgeleitete Klasse (für die unteren Klassen)

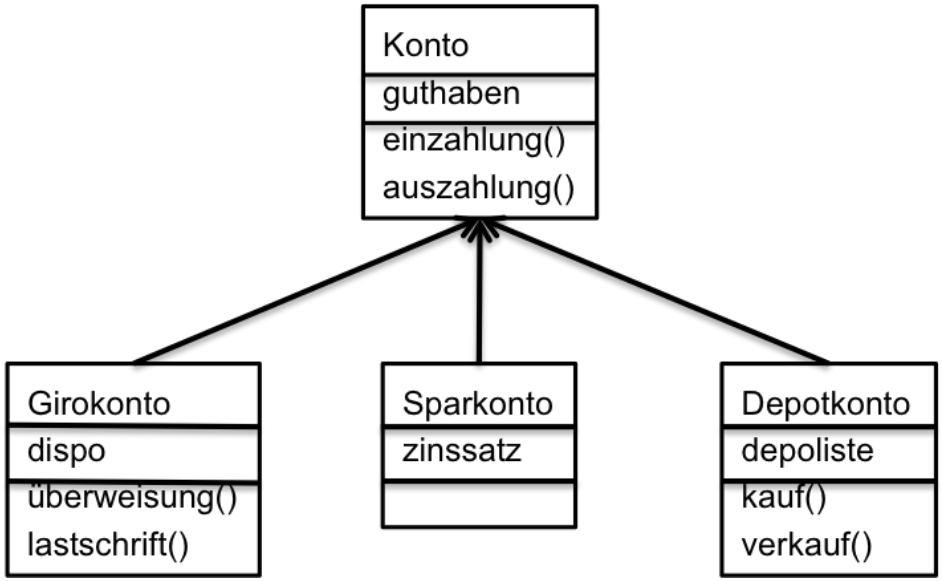
Vererbung

- Unterklassen erben Attribute und Methoden von der Oberklasse



Vererbung

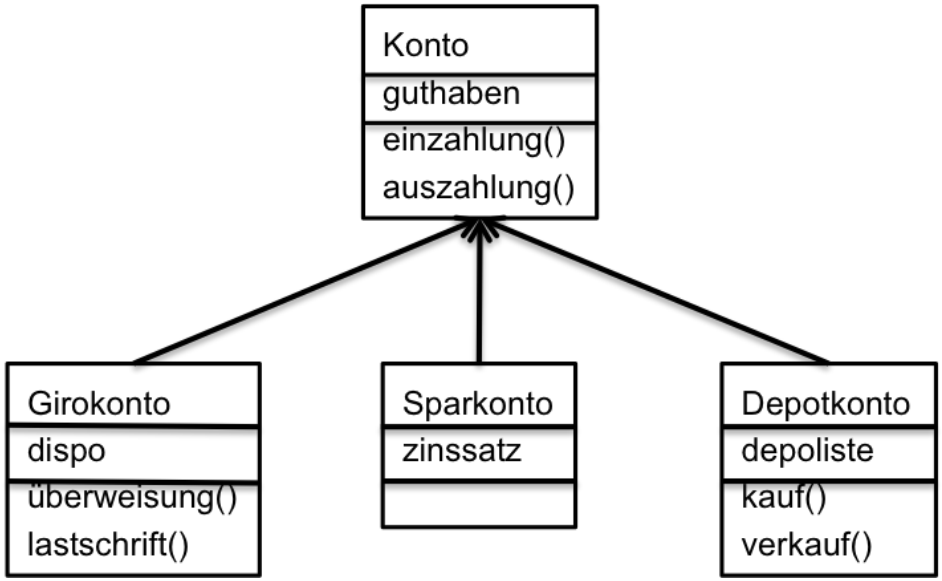
- Unterklassen erben Attribute und Methoden von der Oberklasse



- . . . und können neue Attribute und Methoden einführen

Vererbung

- Unterklassen erben Attribute und Methoden von der Oberklasse



- . . . und können neue Attribute und Methoden einführen
- . . . und können Attribute und Methoden der Oberklasse überschreiben

Vorteile von OOP

UNI FREIBURG

- Abstraktion: Betrachtung der Objekte und ihrer Eigenschaften und Fähigkeiten, ohne Festlegung auf Implementierung

Vorteile von OOP

UNI FREIBURG

- Abstraktion: Betrachtung der Objekte und ihrer Eigenschaften und Fähigkeiten, ohne Festlegung auf Implementierung
- Vererbung: Klarere Struktur und weniger Redundanz

Vorteile von OOP

UNI FREIBURG

- Abstraktion: Betrachtung der Objekte und ihrer Eigenschaften und Fähigkeiten, ohne Festlegung auf Implementierung
- Vererbung: Klarere Struktur und weniger Redundanz
- Datenkapselung: Objekt interagiert nur über vordefinierte Methoden. Implementierung kann verändert werden, ohne dass andere Teile des Programms geändert werden müssen.

Vorteile von OOP

UNI FREIBURG

- Abstraktion: Betrachtung der Objekte und ihrer Eigenschaften und Fähigkeiten, ohne Festlegung auf Implementierung
- Vererbung: Klarere Struktur und weniger Redundanz
- Datenkapselung: Objekt interagiert nur über vordefinierte Methoden. Implementierung kann verändert werden, ohne dass andere Teile des Programms geändert werden müssen.
- Wiederverwendbarkeit: Programme können einfacher erweitert und modifiziert werden. Klassen können auch in anderen Programmen verwendet werden.

Nachteile von OOP

UNI FREIBURG

Bei folgenden Punkten hat der OOP-Ansatz Schwächen:

- Formulierung: Die natürliche Sprache hat keine feste Bindung von Substantiv (entspr. dem Objekt) und Verb (entspr. der Methode).

Nachteile von OOP

UNI FREIBURG

Bei folgenden Punkten hat der OOP-Ansatz Schwächen:

- Formulierung: Die natürliche Sprache hat keine feste Bindung von Substantiv (entspr. dem Objekt) und Verb (entspr. der Methode).
- Klassenhierarchie ist in der realen Welt nicht immer so klar (Kreis-Ellipse-Problem).

Nachteile von OOP

UNI FREIBURG

Bei folgenden Punkten hat der OOP-Ansatz Schwächen:

- Formulierung: Die natürliche Sprache hat keine feste Bindung von Substantiv (entspr. dem Objekt) und Verb (entspr. der Methode).
- Klassenhierarchie ist in der realen Welt nicht immer so klar (Kreis-Ellipse-Problem).
- Transparenz: Kontrollfluss nicht im Quelltext

Nachteile von OOP

UNI FREIBURG

Bei folgenden Punkten hat der OOP-Ansatz Schwächen:

- Formulierung: Die natürliche Sprache hat keine feste Bindung von Substantiv (entspr. dem Objekt) und Verb (entspr. der Methode).
- Klassenhierarchie ist in der realen Welt nicht immer so klar (Kreis-Ellipse-Problem).
- Transparenz: Kontrollfluss nicht im Quelltext
- Ausführungseffizienz: OOP-Anwendungen benötigen häufig mehr Ressourcen (Laufzeit, Speicher, Energie) als prozedurale Formulierungen.

Nachteile von OOP

UNI FREIBURG

Bei folgenden Punkten hat der OOP-Ansatz Schwächen:

- Formulierung: Die natürliche Sprache hat keine feste Bindung von Substantiv (entspr. dem Objekt) und Verb (entspr. der Methode).
- Klassenhierarchie ist in der realen Welt nicht immer so klar (Kreis-Ellipse-Problem).
- Transparenz: Kontrollfluss nicht im Quelltext
- Ausführungseffizienz: OOP-Anwendungen benötigen häufig mehr Ressourcen (Laufzeit, Speicher, Energie) als prozedurale Formulierungen.
- Programmiereffizienz: Kleine Anwendungen sind oft schneller prozedural programmiert.

UNI FREIBURG

OOP: Erste Schritte in Python

Definieren von Klassen

UNI FREIBURG

Python-Interpreter

```
>>> class MyClass:
...     pass # nur notwendig für leere Klasse!
...
>>> MyClass
<class '__main__.MyClass'>
>>> int
<class 'int'>
```

- Neue Klassen werden mit der `class`-Anweisung eingeführt (Konvention: CamelCase-Namen).

Definieren von Klassen

UNI FREIBURG

Python-Interpreter

```
>>> class MyClass:
...     pass # nur notwendig für leere Klasse!
...
>>> MyClass
<class '__main__.MyClass'>
>>> int
<class 'int'>
```

- Neue Klassen werden mit der `class`-Anweisung eingeführt (Konvention: CamelCase-Namen).
  - Beachte: Wie bei Funktionsdefinitionen mit `def` werden die `class`-Anweisung ausgeführt. D.h. wenn man sie in einer bedingten Anweisung unterbringt, werden sie u.U. nicht ausgeführt!

Definieren von Klassen

UNI FREIBURG

Python-Interpreter

```
>>> class MyClass:
...     pass # nur notwendig für leere Klasse!
...
>>> MyClass
<class '__main__.MyClass'>
>>> int
<class 'int'>
```

- Neue Klassen werden mit der `class`-Anweisung eingeführt (Konvention: CamelCase-Namen).
  - Beachte: Wie bei Funktionsdefinitionen mit `def` werden die `class`-Anweisung ausgeführt. D.h. wenn man sie in einer bedingten Anweisung unterbringt, werden sie u.U. nicht ausgeführt!
  - Sehen ähnlich aus wie Typen (und sind tatsächlich solche)

Erzeugung von Instanzen

UNI FREIBURG

- Eine Instanz einer Klasse wird erzeugt, indem man die Klasse wie eine Funktion aufruft.

Erzeugung von Instanzen

UNI FREIBURG

- Eine Instanz einer Klasse wird erzeugt, indem man die Klasse wie eine Funktion aufruft.

Python-Interpreter

```
>>> class MyClass:
...     pass
...
>>> instance1 = MyClass()
>>> instance1
<__main__.MyClass object at 0x101ac51d0>
>>> instance2 = MyClass()
>>> instance1 is instance2
False
>>> instance1 == instance2
False
```

Erzeugung von Instanzen

UNI FREIBURG

- Eine Instanz einer Klasse wird erzeugt, indem man die Klasse wie eine Funktion aufruft.

Python-Interpreter

```
>>> class MyClass:
...     pass
...
>>> instance1 = MyClass()
>>> instance1
<__main__.MyClass object at 0x101ac51d0>
>>> instance2 = MyClass()
>>> instance1 is instance2
False
>>> instance1 == instance2
False
```

- Alle erzeugten Instanzen sind untereinander nicht-identisch und ungleich!

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

- Instanzen verhalten sich wie Records/Strukturen, denen man *dynamisch* neue Attribute zuordnen kann.

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

- Instanzen verhalten sich wie Records/Strukturen, denen man *dynamisch* neue Attribute zuordnen kann.

Python-Interpreter

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> 2 * 3.14 * my_circle.radius
31.4
```

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

- Instanzen verhalten sich wie Records/Strukturen, denen man *dynamisch* neue Attribute zuordnen kann.

Python-Interpreter

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> 2 * 3.14 * my_circle.radius
31.4
```

- D.h. man kann jeder Instanz dynamisch neue Attribute zuordnen – jede Instanz stellt einen eignen Namensraum dar, auf den wir mit der Punktnotation zugreifen.

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

- Instanzen verhalten sich wie Records/Strukturen, denen man *dynamisch* neue Attribute zuordnen kann.

Python-Interpreter

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> 2 * 3.14 * my_circle.radius
31.4
```

- D.h. man kann jeder Instanz dynamisch neue Attribute zuordnen – jede Instanz stellt einen eignen Namensraum dar, auf den wir mit der Punktnotation zugreifen.
- Wie wir für alle Instanzen einer Klasse die selben Attribute erzeugen, sehen wir gleich.

Methoden-Definition

UNI FREIBURG

- Methoden werden als Funktionen innerhalb von Klassen definiert (mit `def`).

Methoden-Definition

UNI FREIBURG

- Methoden werden als Funktionen innerhalb von Klassen definiert (mit `def`).

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle()
```

Methoden-Definition

UNI FREIBURG

- Methoden werden als Funktionen innerhalb von Klassen definiert (mit `def`).

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle()
```

- Den ersten Parameter einer Methode nennt man per Konvention self – dies ist *die Instanz/das Objekt*.

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle(); c.radius = 1
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle(); c.radius = 1
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

  - über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> class Circle:
...     def area(self):
...         return (self.radius * self.radius *
...                 3.14159)
...
>>> c = Circle(); c.radius = 1
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

  - über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder
  - normal über den Instanzen-Namen (dann wird die Instanz implizit übergeben).

Konstruktion von Instanzen

UNI FREIBURG

- Um für alle Instanzen einer Klasse die gleichen Attribute zu haben, werden diese normalerweise in der __init__-Methode eingeführt, die bei der Erzeugung der Instanz aufgerufen wird.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, rad):
...         self.radius = rad
...
>>> circle = Circle(22)
>>> circle.radius
22
>>> circle.radius = 1
>>> circle.radius
1
```

Die `__init__`-Methode

UNI FREIBURG

- Die spezielle Methode mit dem Namen __init__ wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.

Die `__init__`-Methode

UNI FREIBURG

- Die spezielle Methode mit dem Namen `__init__` wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...
>>> circle = Circle(5)
```

Die `__init__`-Methode

UNI FREIBURG

- Die spezielle Methode mit dem Namen __init__ wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...
>>> circle = Circle(5)
```

- Beachte: Alle Attribute sind öffentlich zugreifbar!

Die `__init__`-Methode

UNI FREIBURG

- Die spezielle Methode mit dem Namen `__init__` wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...
>>> circle = Circle(5)
```

- Beachte: Alle Attribute sind öffentlich zugreifbar!
- Beachte: Auch bei Methoden-Definitionen sind benannte und Default-Parameter möglich!

Die `__init__`-Methode

UNI FREIBURG

- Die spezielle Methode mit dem Namen `__init__` wird aufgerufen, wenn die Instanz erzeugt wird. In dieser Methode „erzeugt“ man die Attribute durch Zuweisungen.

Python-Interpreter

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...
>>> circle = Circle(5)
```

- Beachte: Alle Attribute sind öffentlich zugreifbar!
- Beachte: Auch bei Methoden-Definitionen sind benannte und Default-Parameter möglich!
- Beachte: Attributnamen und Parameternamen von Methoden gehören zu verschiedenen Namensräumen.

Ein Kreis ...

UNI FREIBURG

circle.py

```
class Circle:
  def __init__(self, x=0, y=0, radius=1):
    self.x = x
    self.y = y
    self.radius = radius

  def area(self):
    return self.radius * self.radius * 3.14

  def size_change(self, percent):
    self.radius *= (percent / 100)

  def move(self, xchange=0, ychange=0):
   self.x += xchange
   self.y += ychange
```

Kreise bearbeiten

UNI FREIBURG

Python-Interpreter

```
>>> c = Circle(x=1, y=2, radius=5)
>>> c.area()
78.5
>>> c.size_change(50)
>>> c.area()
19.625
>>> c.move(xchange=10, ychange=20)
>>> (c.x, c.y)
(11, 22)
```

UNI FREIBURG

Vererbung

Weitere Objekte

UNI FREIBURG

- Wir wollen jetzt noch andere geometrische Figuren einführen, wie Rechtecke, Quadrate, Dreiecke, Ellipsen, ...

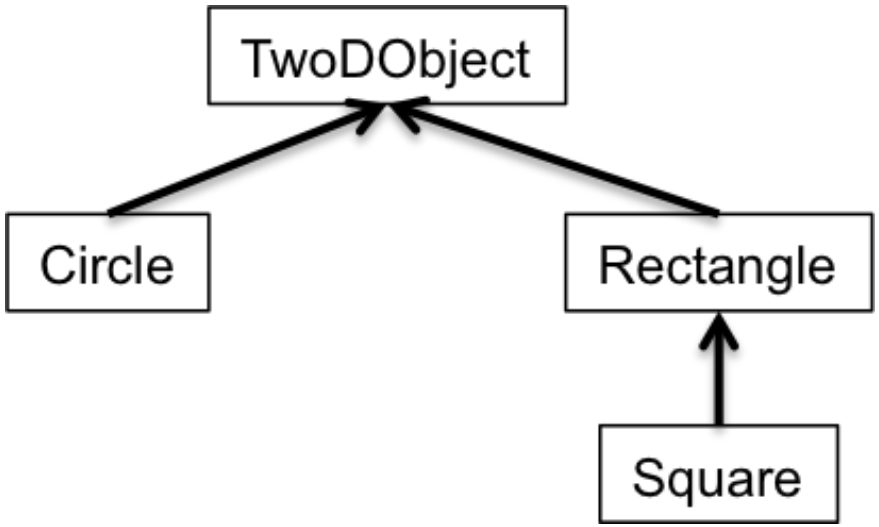
Weitere Objekte

UNI FREIBURG

- Wir wollen jetzt noch andere geometrische Figuren einführen, wie Rechtecke, Quadrate, Dreiecke, Ellipsen, ...
- Diese haben Gemeinsamkeiten (alle haben eine Position in der Ebene) und es gibt Unterschiede (Kreis: Radius, Rechteck: Seiten)

Weitere Objekte

- Wir wollen jetzt noch andere geometrische Figuren einführen, wie Rechtecke, Quadrate, Dreiecke, Ellipsen, ...
- Diese haben Gemeinsamkeiten (alle haben eine Position in der Ebene) und es gibt Unterschiede (Kreis: Radius, Rechteck: Seiten)
- So etwas kann gut in einer Klassenhierarchie dargestellt werden



2D-Objekte

UNI FREIBURG

- Allen Objekten gemeinsam ist, dass sie eine Position in der Ebene haben.

2D-Objekte

UNI FREIBURG

- Allen Objekten gemeinsam ist, dass sie eine Position in der Ebene haben.
- Diese will man ggfs. auch drucken und verändern können.

2D-Objekte

UNI FREIBURG

- Allen Objekten gemeinsam ist, dass sie eine Position in der Ebene haben.
- Diese will man ggfs. auch drucken und verändern können.

geoclasses.py (1)

```
class TwoDObject:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def move(self, xchange=0, ychange=0):
    self.x += xchange
    self.y += ychange

  def print_pos(self):
    print(self.x, self.y)
```

Ein Kreis ist ein 2D-Objekt

UNI FREIBURG

- Jetzt können wir Kreise als eine Spezialisierung von 2D-Objekten einführen und die zusätzlichen und geänderten Attribute und Methoden angeben:

geoclasses.py (2)

```
class Circle(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    self.x = x
    self.y = y

  def area(self):
    return self.radius * self.radius * 3.14

  def size_change(self, percent):
    self.radius *= (percent / 100)
```

Überschreiben versus Erweitern

UNI FREIBURG

- Bei der Vererbung kommen weitere Attribute und Methoden hinzu.

Überschreiben versus Erweitern

UNI FREIBURG

- Bei der Vererbung kommen weitere Attribute und Methoden hinzu.
- Vorhandene Methoden können überschrieben werden (Beispiel: `__init__`).

Überschreiben versus Erweitern

UNI FREIBURG

- Bei der Vererbung kommen weitere Attribute und Methoden hinzu.
- Vorhandene Methoden können überschrieben werden (Beispiel: `__init__`).
- Oft ist es besser, sie zu erweitern – und von der Funktionialität der Methode in der Superklasse Gebrauch zu machen.

Überschreiben versus Erweitern

UNI FREIBURG

- Bei der Vererbung kommen weitere Attribute und Methoden hinzu.
- Vorhandene Methoden können überschrieben werden (Beispiel: `__init__`).
- Oft ist es besser, sie zu erweitern – und von der Funktionialität der Methode in der Superklasse Gebrauch zu machen.

geoclasses.py (3)

```
class Circle1(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    TwoDObject.__init__(self, x, y)
```

Überschreiben versus Erweitern

UNI FREIBURG

- Bei der Vererbung kommen weitere Attribute und Methoden hinzu.
- Vorhandene Methoden können überschrieben werden (Beispiel: `__init__`).
- Oft ist es besser, sie zu erweitern – und von der Funktionialität der Methode in der Superklasse Gebrauch zu machen.

geoclasses.py (3)

```
class Circle1(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    TwoDObject.__init__(self, x, y)
```

- Beachte: Hier wird die Methode über den Klassennamen aufgerufen.

Super!

UNI FREIBURG

- Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.

Super!

UNI FREIBURG

- Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.
- Stattdessen automatisch die Superklasse bestimmen:

Super!

UNI FREIBURG

- Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.
- Stattdessen automatisch die Superklasse bestimmen:

geoclasses.py (4)

```
class Circle2(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    super().__init__(x, y)
```

Super!

UNI FREIBURG

- Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.
- Stattdessen automatisch die Superklasse bestimmen:

geoclasses.py (4)

```
class Circle2(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    super().__init__(x, y)
```

- Beachte: Die Parameterkonventionen müssen bekannt sein oder man muss mit `**kwlist` arbeiten.

Super!

UNI FREIBURG

- Es wird explizit die Methode der aktuellen Superklasse aufgerufen. Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.
- Stattdessen automatisch die Superklasse bestimmen:

geoclasses.py (4)

```
class Circle2(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    super().__init__(x, y)
```

- Beachte: Die Parameterkonventionen müssen bekannt sein oder man muss mit `**kwlist` arbeiten.
- Tatsächlich ist `super()` umstritten: `http://rhettinger.wordpress.com/2011/05/26/super-considered-super/`

Ein Rechteck ist auch ein 2D-Objekt

UNI FREIBURG

- Und weiter geht es mit Rechtecken

Ein Rechteck ist auch ein 2D-Objekt

UNI FREIBURG

- Und weiter geht es mit Rechtecken

geoclasses.py (5)

```
class Rectangle(TwoDObject):
  def __init__(self, x=0, y=0, height=1, width=1):
    self.height = height
    self.width = width
    super().__init__(x, y)

  def area(self):
    return height * width

  def size_change(self, percent):
    self.height *= (percent / 100)
    self.width  *= (percent / 100)
```

Spielen mit 2D-Objekten

UNI FREIBURG

Python-Interpreter

```
>>> t = TwoDObject(10,20)
>>> c = Circle(11,22,5)
>>> r = Rectangle(100,100,20,20)
>>> c.print_pos()
(11,22)
>>> c.move(89,78); c.print_pos()
(100,100)
>>> t.area()
AttributeError: 'TwoDObject' object has no attribute
'area'
>>> r.area()
400
>>> r.size_change(50); r.area()
100
```

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
      super().__init__(x, y, side, side)
```

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
     super().__init__(x, y, side, side)
```

- Was allerdings, wenn wir eine `Square`-Instanz ändern und z.B. dem `height`-Attribut einen neuen Wert zuweisen?

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
        super().__init__(x, y, side, side)
```

- Was allerdings, wenn wir eine `Square`-Instanz ändern und z.B. dem `height`-Attribut einen neuen Wert zuweisen?

→ Die Instanz ist kein Quadrat mehr!

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
     super().__init__(x, y, side, side)
```

- Was allerdings, wenn wir eine `Square`-Instanz ändern und z.B. dem `height`-Attribut einen neuen Wert zuweisen?

→ Die Instanz ist kein Quadrat mehr!

- Allerdings haben wir hier auch auf interne (?) Attribute zugegriffen.

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
      super().__init__(x, y, side, side)
```

- Was allerdings, wenn wir eine `Square`-Instanz ändern und z.B. dem `height`-Attribut einen neuen Wert zuweisen?

→ Die Instanz ist kein Quadrat mehr!

- Allerdings haben wir hier auch auf interne (?) Attribute zugegriffen.
- Was ist mit der Datenkapselung in Python?

Ein Quadrat ist ein Rechteck

UNI FREIBURG

- Ein Quadrat ist ein Spezialfall eines Rechtecks (jedenfalls im mathematischen Sinne).

geoclasses.py (6)

```
class Square(Rectangle):
    def __init__(self, x=0, y=0, side=1):
        super().__init__(x, y, side, side)
```

- Was allerdings, wenn wir eine `Square`-Instanz ändern und z.B. dem `height`-Attribut einen neuen Wert zuweisen?

→ Die Instanz ist kein Quadrat mehr!

- Allerdings haben wir hier auch auf interne (?) Attribute zugegriffen.
- Was ist mit der Datenkapselung in Python?
- Und würde Datenkapselung hier wirklich helfen?

Datenkapselung in Python

UNI FREIBURG

- Die Idee der Datenkapselung ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.

Datenkapselung in Python

UNI FREIBURG

- Die Idee der Datenkapselung ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.
- In anderen OOP-Sprachen existieren Konzepte der Beschränkung wie private (sichtbar nur innerhalb der Klasse), protected (sichtbar in allen Subklassen), public (für jeden sichtbar).

Datenkapselung in Python

UNI FREIBURG

- Die Idee der Datenkapselung ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.
- In anderen OOP-Sprachen existieren Konzepte der Beschränkung wie private (sichtbar nur innerhalb der Klasse), protected (sichtbar in allen Subklassen), public (für jeden sichtbar).
- Python ist da liberal und vertraut darauf, dass die Nutzer vernünftig sind – was das Debuggen z.B. erheblich vereinfacht:

Datenkapselung in Python

UNI FREIBURG

- Die Idee der Datenkapselung ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.
- In anderen OOP-Sprachen existieren Konzepte der Beschränkung wie private (sichtbar nur innerhalb der Klasse), protected (sichtbar in allen Subklassen), public (für jeden sichtbar).
- Python ist da liberal und vertraut darauf, dass die Nutzer vernünftig sind – was das Debuggen z.B. erheblich vereinfacht:
  - Attribute, die nicht mit Unterstrich beginnen, sind für alle sichtbar und modifizierbar.

Datenkapselung in Python

UNI FREIBURG

- Die Idee der Datenkapselung ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.
- In anderen OOP-Sprachen existieren Konzepte der Beschränkung wie private (sichtbar nur innerhalb der Klasse), protected (sichtbar in allen Subklassen), public (für jeden sichtbar).
- Python ist da liberal und vertraut darauf, dass die Nutzer vernünftig sind – was das Debuggen z.B. erheblich vereinfacht:
  - Attribute, die nicht mit Unterstrich beginnen, sind für alle sichtbar und modifizierbar.
  - Attribute, die mit einem Unterstrich beginnen, sind intern und sollten außerhalb nicht benutzt werden.

Datenkapselung in Python

UNI FREIBURG

- Die Idee der Datenkapselung ist, dass die interne Implementation *nicht sichtbar* ist und nur über Methoden zugegriffen wird.
- In anderen OOP-Sprachen existieren Konzepte der Beschränkung wie private (sichtbar nur innerhalb der Klasse), protected (sichtbar in allen Subklassen), public (für jeden sichtbar).
- Python ist da liberal und vertraut darauf, dass die Nutzer vernünftig sind – was das Debuggen z.B. erheblich vereinfacht:
  - Attribute, die nicht mit Unterstrich beginnen, sind für alle sichtbar und modifizierbar.
  - Attribute, die mit einem Unterstrich beginnen, sind intern und sollten außerhalb nicht benutzt werden.
  - Attribute, die mit zwei Unterstrichen beginnen, sind *nicht direkt sichtbar*, da der Klassenname intern mit eingefügt wird (Namens-Massage).

Pythons beschränkt zugreifbare Attribute

UNI FREIBURG

geoclasses.py (7)

```
class TwoDObject1:
  def __init__(self, x=0, y=0):
    self.__x = x
    self._y = y
```

Python-Interpreter

```
>>> td = TwoDObject1(1,2)
>>> td._y
2
>>> td.__x
AttributeError: 'TwoDObject1' object has no attribute
'__x'
>>> td._TwoDObject1__x
1
```

Ein Quadrat ist ein Quadrat ist ein …

UNI FREIBURG

- Ändern wir die Klassendefinitionen so ab, dass alle Instanzenvariablen einen oder zwei Unterstriche als erstes Zeichen haben (also nicht geändert werden sollen), so kann nur die Methode size_change die Attribute ändern.

Ein Quadrat ist ein Quadrat ist ein . . .

UNI FREIBURG

- Ändern wir die Klassendefinitionen so ab, dass alle Instanzenvariablen einen oder zwei Unterstriche als erstes Zeichen haben (also nicht geändert werden sollen), so kann nur die Methode size_change die Attribute ändern.

→ Ein als Quadrat eingeführtes Quadrat bleibt immer Quadrat!

Ein Quadrat ist ein Quadrat ist ein …

UNI FREIBURG

- Ändern wir die Klassendefinitionen so ab, dass alle Instanzenvariablen einen oder zwei Unterstriche als erstes Zeichen haben (also nicht geändert werden sollen), so kann nur die Methode size_change die Attribute ändern.

→ Ein als Quadrat eingeführtes Quadrat bleibt immer Quadrat!

- Was, wenn man *Höhe* und *Breite* separat über Methoden ändern könnte: `stretch_height` und `stretch_width`?

Ein Quadrat ist ein Quadrat ist ein …

UNI FREIBURG

- Ändern wir die Klassendefinitionen so ab, dass alle Instanzenvariablen einen oder zwei Unterstriche als erstes Zeichen haben (also nicht geändert werden sollen), so kann nur die Methode size_change die Attribute ändern.

→ Ein als Quadrat eingeführtes Quadrat bleibt immer Quadrat!

- Was, wenn man *Höhe* und *Breite* separat über Methoden ändern könnte: `stretch_height` und `stretch_width`?

→ Das Kreis-Ellipsen-Problem ist identisch mit dem *Quadrat-Rechteck-Problem*.

Ein Quadrat ist ein Quadrat ist ein …

UNI FREIBURG

- Ändern wir die Klassendefinitionen so ab, dass alle Instanzenvariablen einen oder zwei Unterstriche als erstes Zeichen haben (also nicht geändert werden sollen), so kann nur die Methode size_change die Attribute ändern.

→ Ein als Quadrat eingeführtes Quadrat bleibt immer Quadrat!

- Was, wenn man *Höhe* und *Breite* separat über Methoden ändern könnte: `stretch_height` und `stretch_width`?

→ Das Kreis-Ellipsen-Problem ist identisch mit dem *Quadrat-Rechteck-Problem*.

- Verschiedene Lösungen sind denkbar. M.E. die „vernünftigste“ ist, die beiden Methoden so zu überschreiben, dass jeweils auch der andere Wert geändert wird.

Rechtecke und Quadrate in friedlicher Koexistenz (1)

UNI FREIBURG

geoclasses.py (8)

```
class RectangleStretch(TwoDObject):
  def __init__ ...

  def stretch_height(self, percent):
    self.__height *= (percent / 100.0)

  def stretch_width(self, percent):
    self.__width *= (percent / 100.0)

class SquareStretch(RectangleStretch):
  def __init__ ...

  def stretch_height(self, percent):
    super().stretch_height(percent)
    super().stretch_width(percent)
```

Rechtecke und Quadrate in friedlicher Koexistenz (2)

UNI FREIBURG

- Jetzt wird bei jedem Aufruf von `stretch_height` und `stretch_width` dafür gesorgt, dass die jeweils andere Seite auch geändert wird.

Rechtecke und Quadrate in friedlicher Koexistenz (2)

UNI FREIBURG

- Jetzt wird bei jedem Aufruf von `stretch_height` und `stretch_width` dafür gesorgt, dass die jeweils andere Seite auch geändert wird.
- Beachte: Es kann jetzt auch ein Rechteck geben, das gleiche Höhe und Breite hat! Es ist dann aber nur *zufällig* eine Quadrat. Ein als Quadrat erzeugtes Objekt wird *immer* ein Quadrat sein.

Rechtecke und Quadrate in friedlicher Koexistenz (2)

UNI FREIBURG

- Jetzt wird bei jedem Aufruf von `stretch_height` und `stretch_width` dafür gesorgt, dass die jeweils andere Seite auch geändert wird.
- Beachte: Es kann jetzt auch ein Rechteck geben, das gleiche Höhe und Breite hat! Es ist dann aber nur *zufällig* eine Quadrat. Ein als Quadrat erzeugtes Objekt wird *immer* ein Quadrat sein.
- Alternative Möglichkeit: Eine Instanz *könnte* sich je nachdem, ob die Seiten gleichlang sind oder nicht, dynamisch als Instanz von Rechteck oder Quadrat einordnen.

Rechtecke und Quadrate in friedlicher Koexistenz (2)

UNI FREIBURG

- Jetzt wird bei jedem Aufruf von `stretch_height` und `stretch_width` dafür gesorgt, dass die jeweils andere Seite auch geändert wird.
- Beachte: Es kann jetzt auch ein Rechteck geben, das gleiche Höhe und Breite hat! Es ist dann aber nur *zufällig* eine Quadrat. Ein als Quadrat erzeugtes Objekt wird *immer* ein Quadrat sein.
- Alternative Möglichkeit: Eine Instanz *könnte* sich je nachdem, ob die Seiten gleichlang sind oder nicht, dynamisch als Instanz von Rechteck oder Quadrat einordnen.
- Weitere Alternative: Rechtecke sind Subklassen von Quadraten, da sie mehr Eigenschaften besitzen.

Rechtecke und Quadrate in friedlicher Koexistenz (2)

UNI FREIBURG

- Jetzt wird bei jedem Aufruf von `stretch_height` und `stretch_width` dafür gesorgt, dass die jeweils andere Seite auch geändert wird.
- Beachte: Es kann jetzt auch ein Rechteck geben, das gleiche Höhe und Breite hat! Es ist dann aber nur *zufällig* eine Quadrat. Ein als Quadrat erzeugtes Objekt wird *immer* ein Quadrat sein.
- Alternative Möglichkeit: Eine Instanz *könnte* sich je nachdem, ob die Seiten gleichlang sind oder nicht, dynamisch als Instanz von Rechteck oder Quadrat einordnen.
- Weitere Alternative: Rechtecke sind Subklassen von Quadraten, da sie mehr Eigenschaften besitzen.
- Die Verwirrung ensteht, da die Objekte ja nicht nur statische, unveränderliche Eigenschaften haben, sondern sich ändern: Wie verhält sich ein Quadrat?

Klassenvariablen

UNI FREIBURG

- Können auch Klassen Attribute besitzen?

Klassenvariablen

UNI FREIBURG

- Können auch Klassen Attribute besitzen?

geoclasses.py (9)

```
class TwoDObjectCount:

  counter = 0

  def  __init__(self, x=0, y=0):
    self.x = x
    self.y = y
    TwoDObjectCount.counter += 1
```

Klassenvariablen

UNI FREIBURG

- Können auch Klassen Attribute besitzen?

geoclasses.py (9)

```
class TwoDObjectCount:

  counter = 0

  def  __init__(self, x=0, y=0):
    self.x = x
    self.y = y
    TwoDObjectCount.counter += 1
```

→ Variablen, die innerhalb des Klassenkörpers eingeführt werden, heißen Klassenattribute (oder statische Attribute) und sind (auch) in allen Instanzen (zum Lesen) sichtbar.

Klassenvariablen

UNI FREIBURG

- Können auch Klassen Attribute besitzen?

geoclasses.py (9)

```
class TwoDObjectCount:

  counter = 0

  def  __init__(self, x=0, y=0):
    self.x = x
    self.y = y
    TwoDObjectCount.counter += 1
```

→ Variablen, die innerhalb des Klassenkörpers eingeführt werden, heißen Klassenattribute (oder statische Attribute) und sind (auch) in allen Instanzen (zum Lesen) sichtbar.

- Zum Schreiben müssen sie über den Klassennamen angesprochen werden.

Klassenvariablen: Lesender und schreibender Zugriff

UNI FREIBURG

Python-Interpreter

```
>>> TwoDObjectCount.counter
0
>>> t1 = TwoDObjectCount()
>>> TwoDObjectCount.counter
1
>>> t2 = TwoDObjectCount()
>>> t3 = TwoDObjectCount()
>>> TwoDObjectCount.counter
3
>>> t1.counter
3
>>> t1.counter = 111 # Neues Objekt-Attr. erzeugt!
>>> TwoDObjectCount.counter
3
```

UNI FREIBURG

Ein bisschen GUI

Graphical User Interface

UNI FREIBURG

- Jede moderne Programmiersprache bietet auf den normalen Desktoprechnen heute ein oder mehrere Grahpical User Interfaces an.

Graphical User Interface

UNI FREIBURG

- Jede moderne Programmiersprache bietet auf den normalen Desktoprechnen heute ein oder mehrere Grahpical User Interfaces an.
- In Python gibt es tkinter (integriert), PyGtk, wxWidget, PyQt, . . .

Graphical User Interface

UNI FREIBURG

- Jede moderne Programmiersprache bietet auf den normalen Desktoprechnen heute ein oder mehrere Grahpical User Interfaces an.
- In Python gibt es tkinter (integriert), PyGtk, wxWidget, PyQt, . . .
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.

Graphical User Interface

UNI FREIBURG

- Jede moderne Programmiersprache bietet auf den normalen Desktoprechnen heute ein oder mehrere Grahpical User Interfaces an.
- In Python gibt es tkinter (integriert), PyGtk, wxWidget, PyQt, . . .
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.
- Wir wollen jetzt einen kleinen Teil von `tkinter` kennen lernen, um unsere Geo-Objekte zu visualisieren.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk
import sys

root = tk.Tk()
lab = tk.Label(root, text="Hello World")
lab.pack()

if "idlelib" not in sys.modules:
    root.mainloop()
```

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk
import sys

root = tk.Tk()
lab = tk.Label(root, text="Hello World")
lab.pack()

if "idlelib" not in sys.modules:
    root.mainloop()
```

- `root` wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk
import sys

root = tk.Tk()
lab = tk.Label(root, text="Hello World")
lab.pack()

if "idlelib" not in sys.modules:
    root.mainloop()
```

- `root` wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen.
- `lab` wird ein Label-Widget innerhalb des `root`-Objekts erzeugt.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk
import sys

root = tk.Tk()
lab = tk.Label(root, text="Hello World")
lab.pack()

if "idlelib" not in sys.modules:
    root.mainloop()
```

- `root` wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen.
- `lab` wird ein Label-Widget innerhalb des `root`-Objekts erzeugt.
- Dann wird `lab` in seinem Elternfenster positioniert.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk
import sys

root = tk.Tk()
lab = tk.Label(root, text="Hello World")
lab.pack()

if "idlelib" not in sys.modules:
    root.mainloop()
```

- `root` wird das Wurzelobjekt, in das alle anderen Objekte hineinkommen.
- `lab` wird ein Label-Widget innerhalb des `root`-Objekts erzeugt.
- Dann wird `lab` in seinem Elternfenster positioniert.
- Schließlich wird die Event-Schleife aufgerufen.

IDLE, `tkinter` und die `mainloop()`

UNI FREIBURG

- IDLE selbst ist mit Hilfe von `tkinter` implementiert worden.

IDLE, tkinter und die mainloop()

UNI FREIBURG

- IDLE selbst ist mit Hilfe von `tkinter` implementiert worden.
- Deshalb muss man etwas vorsichtig sein, wenn man `tkinter` in `IDLE` entwickelt.

IDLE, `tkinter` und die `mainloop()`

UNI FREIBURG

- IDLE selbst ist mit Hilfe von `tkinter` implementiert worden.
- Deshalb muss man etwas vorsichtig sein, wenn man `tkinter` in `IDLE` entwickelt.
- Man sollte nicht (noch einmal) `mainloop()` aufrufen (dafür sorgt das if-Statement)

IDLE, tkinter und die mainloop()

UNI FREIBURG

- IDLE selbst ist mit Hilfe von `tkinter` implementiert worden.
- Deshalb muss man etwas vorsichtig sein, wenn man `tkinter` in `IDLE` entwickelt.
- Man sollte nicht (noch einmal) `mainloop()` aufrufen (dafür sorgt das if-Statement)
- Man sollte das Programm nicht beenden, da sonst `tkinter` mit beendet wird.

Objekte malen

UNI FREIBURG

Canvas erzeugen

```
import tkinter as tk
import sys

root = tk.Tk()
cv = tk.Canvas(root, height=600, width=600)
cv.pack()
r1 = cv.create_rectangle(100, 100, 200, 150, fill='green')
o1 = cv.create_oval(400,400,500,500,fill='red',width=3)
```

Objekte malen

UNI FREIBURG

Canvas erzeugen

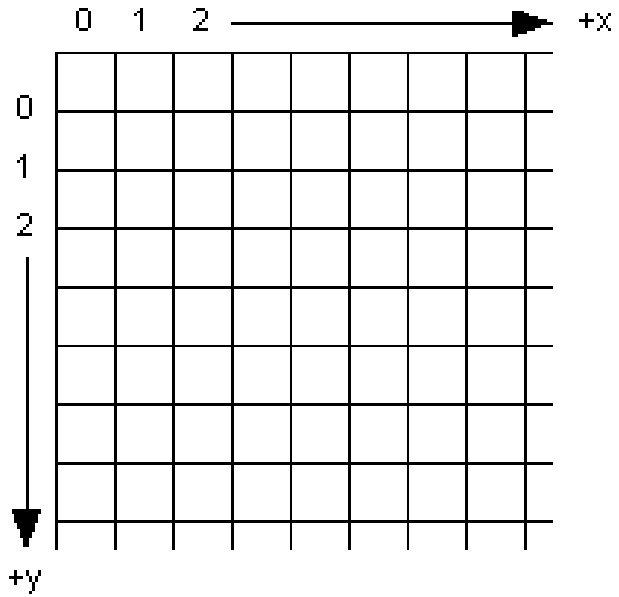
```
import tkinter as tk
import sys

root = tk.Tk()
cv = tk.Canvas(root, height=600, width=600)
cv.pack()
r1 = cv.create_rectangle(100, 100, 200, 150, fill='green')
o1 = cv.create_oval(400,400,500,500,fill='red',width=3)
```

- Ein Canvas ist eine Leinwand, auf der man „malen“ kann.
- Darauf kann man dann verschiedene geometrische Figuren erzeugen.

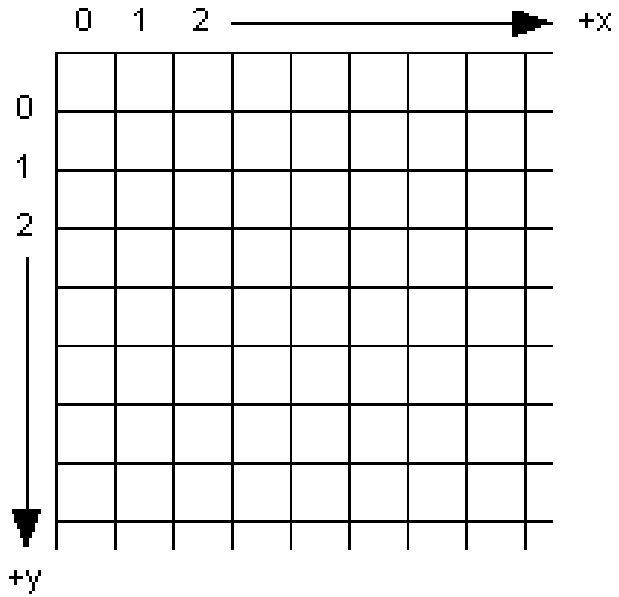
Grafik-Koordinatensysteme

- Im Unterschied zum mathematischen Koordinatensystem liegt der Nullpunkt bei Grafikdarstellungen immer oben links.



Grafik-Koordinatensysteme

- Im Unterschied zum mathematischen Koordinatensystem liegt der Nullpunkt bei Grafikdarstellungen immer oben links.



- Wie gewohnt gibt man (x,y)-Paare zur Bestimmung von Punkten an.

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **kw)`
  zeichnet eine Linie von (x1, y1) nach (x2, y2).

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*`.create_line(x1, y1, x2, y2, **kw)`
  zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*`.create_rectangle(x1, y1, x2, y2, **kw)`
  zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **kw)`
  zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **kw)`
  zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **kw)`
  zeichnet ein Oval innerhalb des Rechtecks geformt durch obere linke Ecke (x1, y1) und untere rechte Ecke (x2, y2).

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **kw)`
  zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **kw)`
  zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **kw)`
  zeichnet ein Oval innerhalb des Rechtecks geformt durch obere linke Ecke (x1, y1) und untere rechte Ecke (x2, y2).
- Alle `create`-Methoden liefern den Index des erzeugten Objekts.

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **kw)`
  zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **kw)`
  zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **kw)`
  zeichnet ein Oval innerhalb des Rechtecks geformt durch obere linke Ecke (x1, y1) und untere rechte Ecke (x2, y2).
- Alle `create`-Methoden liefern den Index des erzeugten Objekts.
- *canvas*.`delete(i)` löscht Objekt mit dem Index *i*.

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **kw)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **kw)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **kw)` zeichnet ein Oval innerhalb des Rechtecks geformt durch obere linke Ecke (x1, y1) und untere rechte Ecke (x2, y2).
- Alle `create`-Methoden liefern den Index des erzeugten Objekts.
- *canvas*.`delete(i)` löscht Objekt mit dem Index *i*.
- *canvas*.`move(i, xdelta, ydelta)` bewegt Objekt um xdelta und ydelta.

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **kw)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **kw)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **kw)` zeichnet ein Oval innerhalb des Rechtecks geformt durch obere linke Ecke (x1, y1) und untere rechte Ecke (x2, y2).
- Alle `create`-Methoden liefern den Index des erzeugten Objekts.
- *canvas*.`delete(i)` löscht Objekt mit dem Index *i*.
- *canvas*.`move(i, xdelta, ydelta)` bewegt Objekt um xdelta und ydelta.
- *canvas*.`update()` erneuert die Darstellung auf dem Bildschirm (auch für andere Fenster möglich).

Visualisierung

UNI FREIBURG

- Wenn wir annehmen, dass die Objektpositionen unserer geometrischen Objekte immer der Schwerpunkt ist, dann könnte man den Kreis wie folgt definieren.

Visualisierung

UNI FREIBURG

- Wenn wir annehmen, dass die Objektpositionen unserer geometrischen Objekte immer der Schwerpunkt ist, dann könnte man den Kreis wie folgt definieren.

Geoclasses visuell

```
class Circle(TwoDObject):
  def __init__(self, x=0, y=0, radius=1):
    self.radius = radius
    super().__init__(x, y)
    self.index = cv.create_oval(self.x-self.radius,
                                self.y-self.radius,
                                self.x+self.radius,
                                self.y+self.radius)
  def move(self, xchange=0, ychange=0):
    self.x += xchange
    self.y += ychange
    cv.move(self.index, xchange, ychange)
```

UNI FREIBURG

Zusammenfassung

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.
- Klassen sind die „Baupläne“ für die Instanzen.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.
- Klassen sind die „Baupläne“ für die Instanzen.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden, und als ersten Parameter immer `self` besitzen sollten.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.
- Klassen sind die „Baupläne“ für die Instanzen.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden, und als ersten Parameter immer `self` besitzen sollten.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.
- Klassen sind die „Baupläne“ für die Instanzen.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden, und als ersten Parameter immer `self` besitzen sollten.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer Vererbungshierarchie angeordnet werden.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein wichtiges **Programmierparadigma**
- Ein Problem wird zerlegt in seine Objekte und die Interaktionen zwischen den Objekten.
- Klassen sind die „Baupläne“ für die Instanzen.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden, und als ersten Parameter immer `self` besitzen sollten.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer Vererbungshierarchie angeordnet werden.
- Es gibt auch Klassenattribute.