Informatik I: Einführung in die Programmierung

12. Objekt-orientierte Programmierung: Einstieg und ein bisschen GUI

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Peter Thiemann

07. Januar 2020

UNI FREIBURG

Motivation

Was ist OOP?

UNI FREIBURG

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

Was ist OOP?

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 OOP?

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 OOP?

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 OOP?

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 OOP?

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

Was ist OOP?

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 kooperierender Objekte

Was ist OOP?

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 kooperierender Objekte
  - Zustand wird mit den Operationen darauf zusammengefasst

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!
- Objekte haben

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!
- Objekte haben
  - in der realen Welt: Zustand und Verhalten

Objekte (im OOP-Sinne)

UNI FREIBURG

- Objekte gibt es im realen Leben überall!
- 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 `balance` 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 `balance` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert.
  Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `withdraw` den Betrag, der unter dem Attribut `balance` 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 `balance` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert.
  Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `withdraw` den Betrag, der unter dem Attribut `balance` 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 `balance` als Zahl gespeichert.
- Verhalten wird durch Methoden realisiert.
  Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `withdraw` den Betrag, der unter dem Attribut `balance` 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;

Klassen und Objekte (1)

UNI FREIBURG

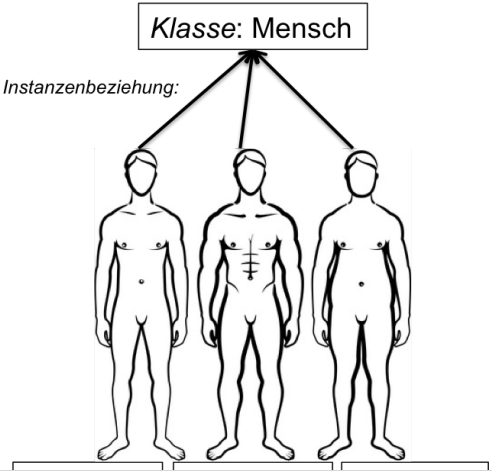
- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
- Ein Objekt / Instanz der Klasse

Klassen und Objekte (1)

UNI FREIBURG

- Eine Klasse
  - ist der „Bauplan“ für bestimmte Objekte;
  - enthält die Definition der Attribute und Methoden;
- Ein Objekt / Instanz der Klasse
  - wird dem „Bauplan“ entsprechend erzeugt

Klassen und Objekte (2)



UNI FREIBURG

OOP: Die nächsten Schritte

Kreis

Ein Kreis wird beschrieben durch seinen Mittelpunkt und seinen Radius.

Wiederholung: Definieren von Klassen

Beispiel: Geometrische Objekte

UNI FREIBURG

Kreis

Ein Kreis wird beschrieben durch seinen Mittelpunkt und seinen Radius.

Klassengerüst

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

Methoden-Definition

UNI FREIBURG

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

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

    def area(self):
        return (self.radius * self.radius * math.pi)
```

Methoden-Definition

UNI FREIBURG

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

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

    def area(self):
        return (self.radius * self.radius * math.pi)
```

- Der erste Parameter einer Methode heißt per Konvention self. Das ist der Empfänger des Methodenaufrufs, d.h. die Instanz, auf der die Methode aufgerufen wird.

Methoden-Aufrufe

- Methoden können aufgerufen werden:

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> c = Circle(1, 0, 0)
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> c = Circle(1, 0, 0)
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

  - zur Not über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> c = Circle(1, 0, 0)
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

  - zur Not über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder
  - über eine Instanz, die dann implizit übergeben wird.

Methoden-Aufrufe

UNI FREIBURG

- Methoden können aufgerufen werden:

Python-Interpreter

```
>>> c = Circle(1, 0, 0)
>>> Circle.area(c)
3.14159
>>> c.area()
3.14159
```

  - zur Not über den Klassennamen (dann muss das `self`-Argument angegeben werden), oder
  - über eine Instanz, die dann implizit übergeben wird.
  - Im *Normalfall* wird `c.area()` verwendet!

Ein Kreis ...

UNI FREIBURG

circle.py

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

    def area(self):
        return self.radius * self.radius * math.pi

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

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

Kreise bearbeiten

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)
```

Weitere geometrische Figuren

- Wir wollen jetzt noch weitere geometrische Figuren einführen, wie Kreissektoren, Rechtecke, Dreiecke, Ellipsen, Kreissegmente, …

Weitere geometrische Figuren

UNI FREIBURG

- Wir wollen jetzt noch weitere geometrische Figuren einführen, wie Kreissektoren, Rechtecke, Dreiecke, Ellipsen, Kreissegmente, ...
- Ein Rechteck wird beschrieben durch seinen Referenzpunkt (links oben) und seine Seitenlängen.

Weitere geometrische Figuren

UNI FREIBURG

- Wir wollen jetzt noch weitere geometrische Figuren einführen, wie Kreissektoren, Rechtecke, Dreiecke, Ellipsen, Kreissegmente, ...
- Ein Rechteck wird beschrieben durch seinen Referenzpunkt (links oben) und seine Seitenlängen.
- Also

Klasse fürs Rechteck

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

  def area(self):
    return self.width * self.height

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

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

Beobachtung

UNI FREIBURG

- Die Bearbeitung des Referenzpunkts (x,y) ist bei `Circle` und `Rectangle` Objekten gleich.

Beobachtung

UNI FREIBURG

- Die Bearbeitung des Referenzpunkts (x,y) ist bei `Circle` und `Rectangle` Objekten gleich.
  - Der Konstruktor behandelt sie gleich.

Beobachtung

UNI FREIBURG

- Die Bearbeitung des Referenzpunkts (x,y) ist bei `Circle` und `Rectangle` Objekten gleich.
  - Der Konstruktor behandelt sie gleich.
  - Die `move` Methode behandelt sie gleich.

Beobachtung

UNI FREIBURG

- Die Bearbeitung des Referenzpunkts (x,y) ist bei `Circle` und `Rectangle` Objekten gleich.
  - Der Konstruktor behandelt sie gleich.
  - Die `move` Methode behandelt sie gleich.
- Gesucht: Ausdrucksmöglichkeit für diese Gemeinsamkeit, sodass der entsprechende Code für den `move` und die Initialisierung der Attribute nur einmal geschrieben werden muss.

Beobachtung

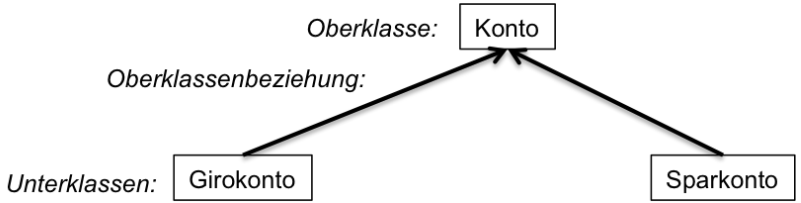
UNI FREIBURG

- Die Bearbeitung des Referenzpunkts (x,y) ist bei `Circle` und `Rectangle` Objekten gleich.
  - Der Konstruktor behandelt sie gleich.
  - Die `move` Methode behandelt sie gleich.
- Gesucht: Ausdrucksmöglichkeit für diese Gemeinsamkeit, sodass der entsprechende Code für den `move` und die Initialisierung der Attribute nur einmal geschrieben werden muss.

$\Rightarrow$ Ein neues Verfahren zur Abstraktion: Vererbung!

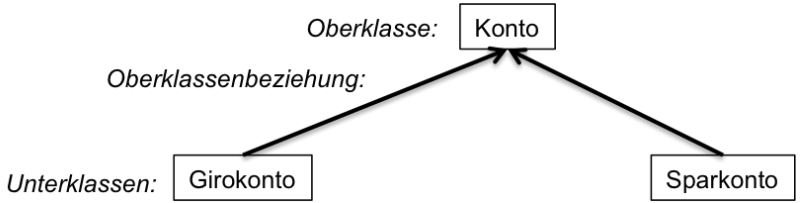
Vererbung

Klassenhierarchien



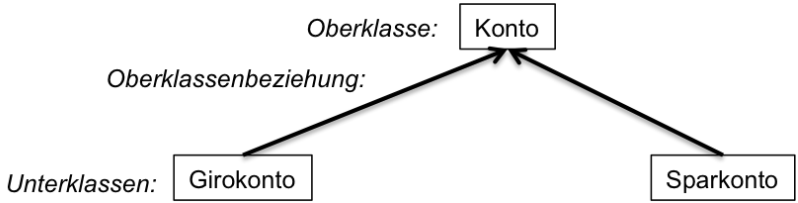
- Verschiedene Arten von Klassen können oft in einer Generalisierungshierarchie (Vererbungshierarchie) angeordnet werden.

Klassenhierarchien



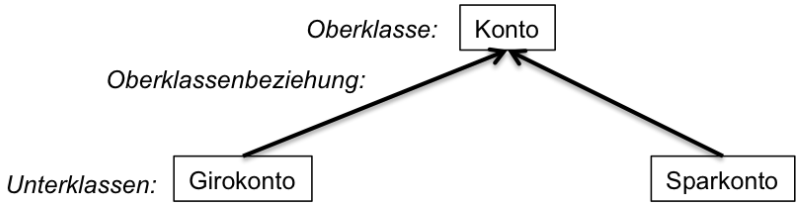
- Verschiedene Arten von Klassen können oft in einer Generalisierungshierarchie (Vererbungshierarchie) angeordnet werden.
- Terminologie:

Klassenhierarchien



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

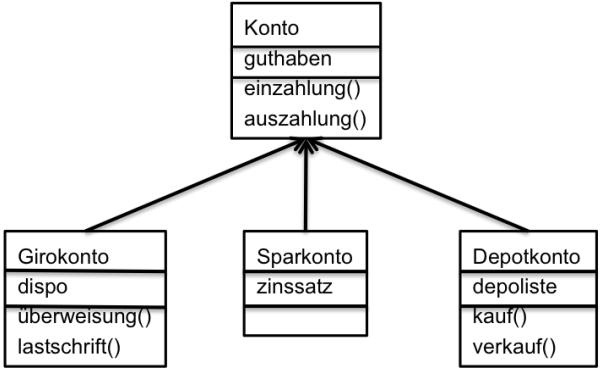
Klassenhierarchien



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

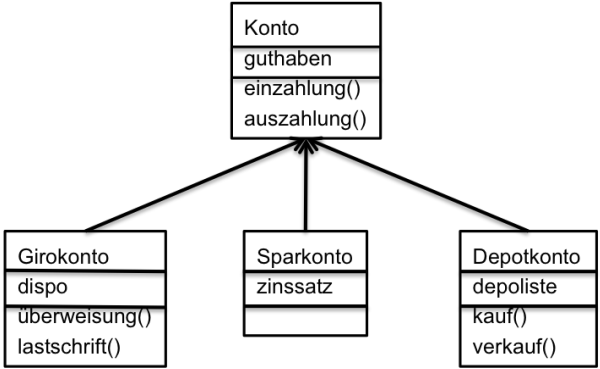
Vererbung

- Unterklassen erben Attribute und Methoden von der Oberklasse



Vererbung

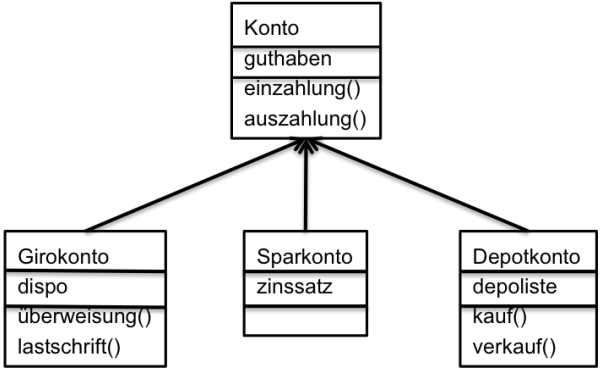
- Unterklassen erben Attribute und Methoden von der Oberklasse



- . . . und können zusätzlich neue Attribute und Methoden einführen

Vererbung

- Unterklassen erben Attribute und Methoden von der Oberklasse



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

Vererbung konkret

Weitere geometrische Objekte

UNI FREIBURG

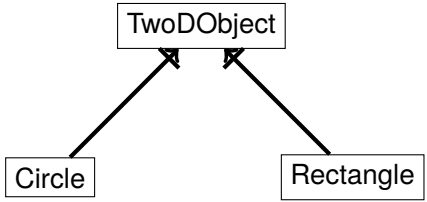
- Fasse die Gemeinsamkeiten der Klassen (alle haben einen Referenzpunkt, der verschoben werden kann) in einer eigenen Klasse zusammen.

Weitere geometrische Objekte

- Fasse die Gemeinsamkeiten der Klassen (alle haben einen Referenzpunkt, der verschoben werden kann) in einer eigenen Klasse zusammen.
- Die Unterschiede werden in spezialisierten Subklassen implementiert.

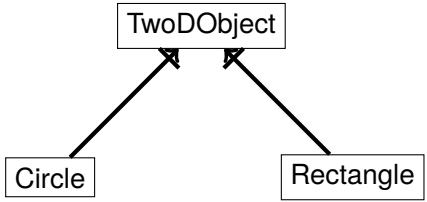
Weitere geometrische Objekte

- Fasse die Gemeinsamkeiten der Klassen (alle haben einen Referenzpunkt, der verschoben werden kann) in einer eigenen Klasse zusammen.
- Die Unterschiede werden in spezialisierten Subklassen implementiert.
- Daraus ergibt sich eine Klassenhierarchie:



Weitere geometrische Objekte

- Fasse die Gemeinsamkeiten der Klassen (alle haben einen Referenzpunkt, der verschoben werden kann) in einer eigenen Klasse zusammen.
- Die Unterschiede werden in spezialisierten Subklassen implementiert.
- Daraus ergibt sich eine Klassenhierarchie:

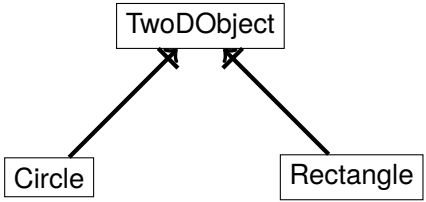


- `TwoDObject` ist Superklasse von `Circle` und `Rectangle`.

Weitere geometrische Objekte

UNI FREIBURG

- Fasse die Gemeinsamkeiten der Klassen (alle haben einen Referenzpunkt, der verschoben werden kann) in einer eigenen Klasse zusammen.
- Die Unterschiede werden in spezialisierten Subklassen implementiert.
- Daraus ergibt sich eine Klassenhierarchie:



- `TwoDObject` ist Superklasse von `Circle` und `Rectangle`.
- `Circle` und `Rectangle` sind Subklassen von `TwoDObject`.

2D-Objekte

UNI FREIBURG

- Alle geometrischen Figuren besitzen einen Referenzpunkt, der verschoben werden kann, und sie besitzen eine Fläche.

2D-Objekte

UNI FREIBURG

- Alle geometrischen Figuren besitzen einen Referenzpunkt, der verschoben werden kann, und sie besitzen eine Fläche.

geoclasses.py (1)

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

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

    def area(self):
        return 0
```

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, radius:float=1, x:float=0, y:float=0):
    self.radius = radius
    self.x = x
    self.y = y

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

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

Überschreiben (*override*)

UNI FREIBURG

- Durch Vererbung kommen weitere Attribute (hier: `radius`) und Methoden hinzu (hier: `size_change` ist neu; `move` und `area` werden von der Superklasse `TwoDObject` geerbt).

Überschreiben (*override*)

UNI FREIBURG

- Durch Vererbung kommen weitere Attribute (hier: `radius`) und Methoden hinzu (hier: `size_change` ist neu; `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Geerbte Methoden können überschrieben werden (Beispiel: `area`), indem wir in der Subklasse eine neue Definition angeben.

Überschreiben (*override*)

UNI FREIBURG

- Durch Vererbung kommen weitere Attribute (hier: `radius`) und Methoden hinzu (hier: `size_change` ist neu; `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Geerbte Methoden können überschrieben werden (Beispiel: `area`), indem wir in der Subklasse eine neue Definition angeben.
- Auf einer `Circle` Instanz wird aufgerufen

Überschreiben (*override*)

UNI FREIBURG

- Durch Vererbung kommen weitere Attribute (hier: `radius`) und Methoden hinzu (hier: `size_change` ist neu; `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Geerbte Methoden können überschrieben werden (Beispiel: `area`), indem wir in der Subklasse eine neue Definition angeben.
- Auf einer `Circle` Instanz wird aufgerufen
  - `move` aus `TwoDObject`

Überschreiben (*override*)

UNI FREIBURG

- Durch Vererbung kommen weitere Attribute (hier: `radius`) und Methoden hinzu (hier: `size_change` ist neu; `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Geerbte Methoden können überschrieben werden (Beispiel: `area`), indem wir in der Subklasse eine neue Definition angeben.
- Auf einer `Circle` Instanz wird aufgerufen
  - `move` aus `TwoDObject`
  - `area` aus `Circle`

Überschreiben (*override*)

- Durch Vererbung kommen weitere Attribute (hier: `radius`) und Methoden hinzu (hier: `size_change` ist neu; `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Geerbte Methoden können überschreiben werden (Beispiel: `area`), indem wir in der Subklasse eine neue Definition angeben.
- Auf einer `Circle` Instanz wird aufgerufen
  - `move` aus `TwoDObject`
  - `area` aus `Circle`
  - `__init__` aus `Circle`

Dynamische Bindung (Dynamic Dispatch)

- Das Verhalten eines Methodenaufrufs wie `obj.area()` bzw `obj.move()` wird erst zur Laufzeit des Programms bestimmt.
- Es hängt ab vom (Laufzeit-) Typ von `obj`.
  - Falls `obj` eine Instanz von `TwoDObject` ist, also falls `type(obj) == TwoDObject`, dann wird sowohl für `area` als auch für `move` der Code aus `TwoDObject` verwendet.
  - Falls `obj` eine Instanz von `Circle` ist, also falls `type(obj) == Circle`, dann wird für `area` der Code aus `Circle` und für `move` der Code aus `TwoDObject` verwendet.
- Dieses Verhalten heißt dynamische Bindung bzw dynamic dispatch und ist eine definierende Eigenschaft für objekt-orientierte Sprachen.

Beispiel

Python-Interpreter

```
>>> t = TwoDObject(x=10, y=20)
>>> t.area()
0
>>> t.move(xchange=10, ychange=20)
>>> t.x, t.y
(20, 40)
>>> t.size_change(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'TwoDObject' object has no attribute 'size_change'
>>> 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)
```

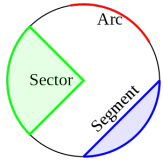
Kreissektoren

UNI FREIBURG

- Ein Kreissektor wird beschrieben durch einen Kreis und einen Winkel:

Kreissektoren

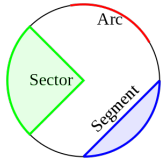
- Ein Kreissektor wird beschrieben durch einen Kreis und einen Winkel:



https://commons.wikimedia.org/wiki/File:Circle_slices.svg (public domain)

Kreissektoren

- Ein Kreissektor wird beschrieben durch einen Kreis und einen Winkel:



`https://commons.wikimedia.org/wiki/File:Circle_slices.svg` (public domain)

- Für Sektoren können wir eine Subklasse von `Circle` anlegen.

Kreissektor als Subklasse vom Kreis

UNI FREIBURG

```
class Sector (Circle):
    def __init__(self, angle:float= 180, radius:float=1, x:float=0, y:float=0):
        self.angle = angle
        self.radius = radius
        self.x = x
        self.y = y

    def area(self):
        return self.radius * self.radius * math.pi * self.angle / 360
```

Kreissektor als Subklasse vom Kreis

UNI FREIBURG

```
class Sector (Circle):
    def __init__(self, angle:float= 180, radius:float=1, x:float=0, y:float=0):
        self.angle = angle
        self.radius = radius
        self.x = x
        self.y = y

    def area(self):
        return self.radius * self.radius * math.pi * self.angle / 360
```

Sector verwendet

- `move` von `TwoDObject`
- `size_change` von `Circle`
- `area` von `Sector`, aber ein Teil des Codes ist aus `Circle` kopiert!

Super!

UNI FREIBURG

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?

Super!

UNI FREIBURG

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Die überschriebene Methode der Superklassekönnte explizit über den Klassennamen aufgerufen werden.

Super!

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Die überschriebene Methode der Superklassekönnte explizit über den Klassennamen aufgerufen werden.

Expliziter Aufruf (fehleranfällig!)

```
class Sector1(Circle):
    ...
    def area(self):
        return Circle.area(self) * self.angle / 360
```

Super!

UNI FREIBURG

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Die überschriebene Methode der Superklassekönnte explizit über den Klassennamen aufgerufen werden.

Expliziter Aufruf (fehleranfällig!)

```
class Sector1(Circle):
    ...
    def area(self):
        return Circle.area(self) * self.angle / 360
```

- Fehlerquelle: Wenn sich die Hierarchie ändert (z.B. auch nur der Name der Superklasse), muss beim Methodenaufruf nachgebessert werden.

Super!

- Python kann die Superklasse automatisch bestimmen!

Super!

- Python kann die Superklasse automatisch bestimmen!

Verwendung von super (besser)

```
class Sector1(Circle):
    ...
    def area(self):
        return super().area() * self.angle / 360
```

Super!

- Python kann die Superklasse automatisch bestimmen!

Verwendung von super (besser)

```
class Sector1(Circle):
    ...
    def area(self):
        return super().area() * self.angle / 360
```

- `super()` nur innerhalb von Methoden verwenden.

Super!

UNI FREIBURG

- Python kann die Superklasse automatisch bestimmen!

Verwendung von super (besser)

```
class Sector1(Circle):
    ...
    def area(self):
        return super().area() * self.angle / 360
```

- `super()` nur innerhalb von Methoden verwenden.
- `super().method(...)` ruft `method` auf dem Empfänger (also `self`) auf, aber tut dabei so, als ob `self` Instanz der Superklasse wäre.

Super!

UNI FREIBURG

- Python kann die Superklasse automatisch bestimmen!

Verwendung von super (besser)

```
class Sector1(Circle):
    ...
    def area(self):
        return super().area() * self.angle / 360
```

- `super()` nur innerhalb von Methoden verwenden.
- `super().method(...)` ruft `method` auf dem Empfänger (also `self`) auf, aber tut dabei so, als ob `self` Instanz der Superklasse wäre.
- Im Beispiel wird `area` in `Circle` aufgerufen.

Beispiel

UNI FREIBURG

Python-Interpreter

```
>>> s = Sector (x=1, y=2, radius=5, angle=90)
>>> s.area()
19.634954084936208
>>> c = Circle (x=1, y=2, radius=5)
>>> c.area()
78.53981633974483
>>> assert s.area() * 4 == c.area()
>>> s.move(9,8)
>>> s.x, s.y
(10, 10)
>>> s.size_change(200)
>>> s.area()
78.53981633974483
```

Die `__init__` Methode

- `__init__` wird ebenfalls in `Circle` und `Sector` überschrieben.

Die `__init__` Methode

- `__init__` wird ebenfalls in `Circle` und `Sector` überschrieben.
- Auch hier kann `super` verwendet werden.

Die `__init__` Methode

- `__init__` wird ebenfalls in `Circle` und `Sector` überschrieben.
- Auch hier kann `super` verwendet werden.

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

class Sector2(Circle2):
    def __init__(self, angle:float= 180, radius:float=1, x:float=0, y:float=0):
        self.angle = angle
        super().__init__(radius, x, y)
    ...
```

Praxistipp: keyword parameter

UNI FREIBURG

- Falls die `__init__` Methode der Superklasse viele Parameter hat oder sich diese Parameter im Laufe der Zeit ändern können, dann ist es unschön, wenn diese Parameter im Code der Subklasse festgelegt werden.

Praxistipp: keyword parameter

- Falls die `__init__` Methode der Superklasse viele Parameter hat oder sich diese Parameter im Laufe der Zeit ändern können, dann ist es unschön, wenn diese Parameter im Code der Subklasse festgelegt werden.
- In Python gibt es die Möglichkeit, neben festen Parameterlisten auch beliebige Parameterlisten zu definieren, wobei die einzelnen Parameter durch ihren Namen (Schlüsselwort) angewählt werden können.

Praxistipp: keyword parameter

- Falls die `__init__` Methode der Superklasse viele Parameter hat oder sich diese Parameter im Laufe der Zeit ändern können, dann ist es unschön, wenn diese Parameter im Code der Subklasse festgelegt werden.
- In Python gibt es die Möglichkeit, neben festen Parameterlisten auch beliebige Parameterlisten zu definieren, wobei die einzelnen Parameter durch ihren Namen (Schlüsselwort) angewählt werden können.
- Daher der Name keyword parameter.

Praxistipp: keyword parameter

UNI FREIBURG

- Falls die `__init__` Methode der Superklasse viele Parameter hat oder sich diese Parameter im Laufe der Zeit ändern können, dann ist es unschön, wenn diese Parameter im Code der Subklasse festgelegt werden.
- In Python gibt es die Möglichkeit, neben festen Parameterlisten auch beliebige Parameterlisten zu definieren, wobei die einzelnen Parameter durch ihren Namen (Schlüsselwort) angewählt werden können.
- Daher der Name keyword parameter.
- Zum Abfangen aller nicht genannten keyword parameter gibt es eine spezielle Syntax `**kwargs`, die sowohl in der formalen Parameterliste, wie auch im Funktionsaufruf verwendet werden kann.

Praxistipp: keyword parameter

UNI FREIBURG

- Falls die `__init__` Methode der Superklasse viele Parameter hat oder sich diese Parameter im Laufe der Zeit ändern können, dann ist es unschön, wenn diese Parameter im Code der Subklasse festgelegt werden.
- In Python gibt es die Möglichkeit, neben festen Parameterlisten auch beliebige Parameterlisten zu definieren, wobei die einzelnen Parameter durch ihren Namen (Schlüsselwort) angewählt werden können.
- Daher der Name keyword parameter.
- Zum Abfangen aller nicht genannten keyword parameter gibt es eine spezielle Syntax `**kwargs`, die sowohl in der formalen Parameterliste, wie auch im Funktionsaufruf verwendet werden kann.
- Hier und jetzt nur die Verwendung in `__init__`, Details später im Zusammenhang mit dem Datentyp Dictionary.

Praxistipp: `__init__` mit keyword parameter

```
class Circle3(TwoDObject):
    def __init__(self, radius:float=1, **kwargs):
        self.radius = radius
        super().__init__(**kwargs)
    ...
class Sector3(Circle3):
    def __init__(self, angle:float= 180, **kwargs):
        self.angle = angle
        super().__init__(**kwargs)
    ...
```

- Beachte:
  - jede Klasse benennt nur die Argumente des Konstruktors `__init__`, die lokal verwendet werden.
  - der Aufruf von `super().__init__(**kwargs)` ist immer gleich!
- Anmerkung: `**kwargs` lässt sich mit jeder Funktion verwenden, aber nur an letzter Position der Parameterliste.

Ein Rechteck ist auch ein 2D-Objekt

- Und weiter geht es mit Rechtecken

Ein Rechteck ist auch ein 2D-Objekt

- Und weiter geht es mit Rechtecken

geoclasses.py (5)

```
class Rectangle(TwoDObject):
  def __init__(self, height:float=1, width:float=1, **kwargs):
    self.height = height
    self.width = width
    super().__init__(**kwargs)

  def area(self):
    return self.height * self.width

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

Spielen mit 2D-Objekten

UNI FREIBURG

Python-Interpreter

```
>>> c = Circle(5,11,22)
>>> r = Rectangle(100,100,20,20)
>>> c.x,c.y
(11,22)
>>> c.move(89,78); c.x,c.y
(100,100)
>>> t.area()
0
>>> r.area()
400
>>> r.size_change(50); r.area()
100
```

Die `__str__` Methode

UNI FREIBURG

- Wenn ein Objekt eine Methode namens `__str__` besitzt, dann muss sie eine Funktion definieren, die das Objekt in einen String umwandelt.

Die `__str__` Methode

UNI FREIBURG

- Wenn ein Objekt eine Methode namens `__str__` besitzt, dann muss sie eine Funktion definieren, die das Objekt in einen String umwandelt.
- Sie wird automagisch von der `str` Funktion aufgerufen.

Die `__str__` Methode

UNI FREIBURG

- Wenn ein Objekt eine Methode namens `__str__` besitzt, dann muss sie eine Funktion definieren, die das Objekt in einen String umwandelt.
- Sie wird automagisch von der `str` Funktion aufgerufen.

Aufgabe

Definiere eine `__str__` Methode in `TwoDObject`, die auch für alle Subklassen korrekt funktioniert.
Das heißt, sie liefert den Namen der Klasse, mit der die Instanz konstruiert wurde, und sämtliche Konstruktorparameter mit ihren Namen.

Gerüst von `__str__`

UNI FREIBURG

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

  def __str__ (self):
    n = self.getName()
    p = self.getParameters()
    return n + "(" + p[1:] + ")"
```

Gerüst von `__str__`

UNI FREIBURG

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

  def __str__ (self):
    n = self.getName()
    p = self.getParameters()
    return n + "(" + p[1:] + ")"
```

Wunschdenken

- Die Methode `getName` liefere jeweils den Namen des Konstruktors.
- Die Methode `getParameters` liefere jeweils die Argumentliste als String.

Die Basisklasse

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

  def __str__ (self):
    n = self.getName()
    p = self.getParameters()
    return n + "(" + p[1:] + ")"

  def getName (self) -> str:
    return "TwoDObject"

  def getParameters (self) -> str:
    return ",x=" + repr (self.x) + ",y=" + repr (self.y)
```

Die Basisklasse

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

  def __str__ (self):
    n = self.getName()
    p = self.getParameters()
    return n + "(" + p[1:] + ")"

  def getName (self) -> str:
    return "TwoDObject"

  def getParameters (self) -> str:
    return ",x=" + repr (self.x) + ",y=" + repr (self.y)
```

- In den Subklassen muss `getName` und `getParameters` überschrieben werden

Die Subklassen

UNI FREIBURG

```
class CircleS(TwoDObjectS):
  def __init__(self, radius:float=1, **kwargs):
    self.radius = radius
    super().__init__ (**kwargs)

  def getName (self) -> str:
    return "Circle"

  def getParameters (self) -> str:
    return ",radius=" + repr(self.radius) + super().getParameters()
```

- der eigene Parameter zuerst, dann die Parameter der Superklasse durch den Aufruf von `super().getParameters()`

Die Subklasse der Subklasse

```
class SectorS(CircleS):
  def __init__(self, angle:float=180, **kwargs):
    self.angle = angle
    super().__init__ (**kwargs)

  def getName (self) -> str:
    return "Sector"

  def getParameters (self) -> str:
    return ",angle=" + repr (self.angle) + super().getParameters()
```

- analog zu `CircleS`
- `RectangleS` würde genauso aussehen, aber natürlich mit anderen Strings (selbst!)

UNI FREIBURG

Vermischtes

Klassenvariablen

UNI FREIBURG

- Auch Klassen können Attribute besitzen!

Klassenvariablen

UNI FREIBURG

- Auch Klassen können Attribute besitzen!

geoclasses.py (9)

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

Klassenvariablen

UNI FREIBURG

- Auch Klassen können Attribute besitzen!

geoclasses.py (9)

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

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

Klassenvariablen

UNI FREIBURG

- Auch Klassen können Attribute besitzen!

geoclasses.py (9)

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

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

- Zum Schreiben müssen sie über den Klassennamen angesprochen werden.

Klassenvariablen: Lesender und schreibender Zugriff

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
```

Ein bisschen GUI

Graphical User Interface

UNI FREIBURG

- Jede moderne Programmiersprache bietet heute eine oder mehrere APIs (Application Programming Interface) für GUIs (Graphical User Interface) an.

Graphical User Interface

UNI FREIBURG

- Jede moderne Programmiersprache bietet heute eine oder mehrere APIs (Application Programming Interface) für GUIs (Graphical User Interface) an.
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.

Graphical User Interface

- Jede moderne Programmiersprache bietet heute eine oder mehrere APIs (Application Programming Interface) für GUIs (Graphical User Interface) an.
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.
- In Python gibt es tkinter (integriert), PyGtk, wxWidget, PyQt, uvam.

Graphical User Interface

- Jede moderne Programmiersprache bietet heute eine oder mehrere APIs (Application Programming Interface) für GUIs (Graphical User Interface) an.
- Möglichkeit per Fenster und Mausinteraktion zu interagieren.
- In Python gibt es tkinter (integriert), PyGtk, wxWidget, PyQt, uvam.
- 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

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

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte durch einen Baum von Widgets

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte durch einen Baum von Widgets
- Ein Widget ist eine (rechteckige) Fläche auf dem Bildschirm, auf der eine bestimmte Funktionalität implementiert ist.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte durch einen Baum von Widgets
- Ein Widget ist eine (rechteckige) Fläche auf dem Bildschirm, auf der eine bestimmte Funktionalität implementiert ist.
- `root` wird das Wurzelobjekt, in das alle anderen Widgets aufnimmt.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte durch einen Baum von Widgets
- Ein Widget ist eine (rechteckige) Fläche auf dem Bildschirm, auf der eine bestimmte Funktionalität implementiert ist.
- `root` wird das Wurzelobjekt, in das alle anderen Widgets aufnimmt.
- `lab` wird als Label-Widget an das `root`-Objekt angehängt.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte durch einen Baum von Widgets
- Ein Widget ist eine (rechteckige) Fläche auf dem Bildschirm, auf der eine bestimmte Funktionalität implementiert ist.
- `root` wird das Wurzelobjekt, in das alle anderen Widgets aufnimmt.
- `lab` wird als Label-Widget an das `root`-Objekt angehängt.
- Das Label-Widget kann nur einen String als Text anzeigen.

Hello World

UNI FREIBURG

Hello World

```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte durch einen Baum von Widgets
- Ein Widget ist eine (rechteckige) Fläche auf dem Bildschirm, auf der eine bestimmte Funktionalität implementiert ist.
- `root` wird das Wurzelobjekt, in das alle anderen Widgets aufnimmt.
- `lab` wird als Label-Widget an das `root`-Objekt angehängt.
- Das Label-Widget kann nur einen String als Text anzeigen.
- Dann wird `lab` in seinem Elternfenster positioniert.

Objekte malen

UNI FREIBURG

Canvas erzeugen

```
import tkinter as tk

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)
```

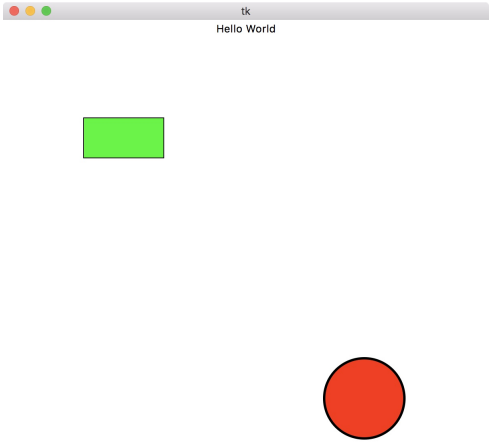
Objekte malen

UNI FREIBURG

Canvas erzeugen

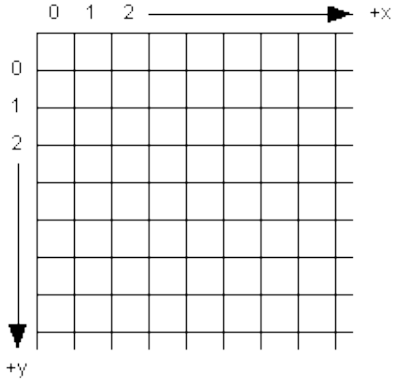
```
import tkinter as tk

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 ein Widget, das wie eine Leinwand funktioniert, auf der verschiedene geometrische Figuren gemalt werden können.



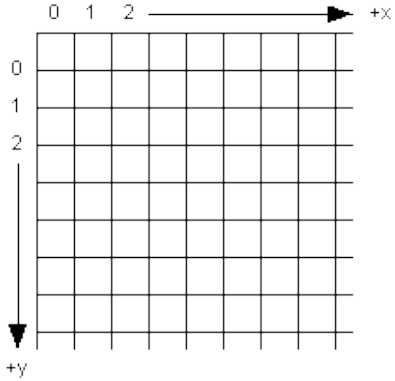
Grafik-Koordinatensysteme

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



Grafik-Koordinatensysteme

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



Einige Canvas-Methoden

UNI FREIBURG

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

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **options)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **options)` 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, **options)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **options)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **options)` 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, **options)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **options)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **options)` 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, **options)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **options)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **options)` 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 das Objekt mit dem Index *i*.

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **options)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **options)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **options)` 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 das Objekt mit dem Index *i*.
- *canvas*.`move(i, xdelta, ydelta)` bewegt Objekt *i* um `xdelta` und `ydelta`.

Einige Canvas-Methoden

UNI FREIBURG

- *canvas*.`create_line(x1, y1, x2, y2, **options)` zeichnet eine Linie von (x1, y1) nach (x2, y2).
- *canvas*.`create_rectangle(x1, y1, x2, y2, **options)` zeichnet ein Rechteck mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- *canvas*.`create_oval(x1, y1, x2, y2, **options)` 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 das Objekt mit dem Index *i*.
- *canvas*.`move(i, xdelta, ydelta)` bewegt Objekt *i* um `xdelta` und `ydelta`.
- *canvas*.`update()` erneuert die Darstellung auf dem Bildschirm.

Visualisierung

UNI FREIBURG

Geoclasses visuell

```
class TwoDObjectV:
    def __init__(self, cv=None, x=0, y=0):
        self.x = x
        self.y = y
        self.cv = cv
        self.index = 0

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

Visualisierung

UNI FREIBURG

Geoclasses visuell

```
class CircleV(TwoDObjectV):
    def __init__(self, radius=1, **kwargs):
        self.radius = radius
        super().__init__(**kwargs)

    def draw(self):
        self.index = self.cv.create_oval(self.x-self.radius,
                                         self.y-self.radius,
                                         self.x+self.radius,
                                         self.y+self.radius)
```

UNI FREIBURG

Zusammenfassung

Zusammenfassung

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

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- Klassen sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- Klassen sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden. Der erste Parameter ist immer `self`, das Empfängerobjekt.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- Klassen sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden. Der erste Parameter ist immer `self`, das Empfängerobjekt.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- Klassen sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden. Der erste Parameter ist immer `self`, das Empfängerobjekt.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer Vererbungshierarchie angeordnet werden.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- Klassen sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden. Der erste Parameter ist immer `self`, das Empfängerobjekt.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer Vererbungshierarchie angeordnet werden.
- Subklassen erben Methoden und Attribute der Superklassen; Methoden der Superklassen können überschrieben werden.

Zusammenfassung

UNI FREIBURG

- Objekt-orientierte Programmierung ist ein **Programmierparadigma**
- Ein Objekt fasst Zustand (Attribute) und die Operationen darauf (Methoden) zusammen.
- Klassen sind die „Baupläne“ für die Objekte. Sie definieren Attribute und Methoden.
- Methoden sind Funktionen, die innerhalb der Klasse definiert werden. Der erste Parameter ist immer `self`, das Empfängerobjekt.
- Attribute werden innerhalb der `__init__`-Methode initialisiert.
- Klassen können in einer Vererbungshierarchie angeordnet werden.
- Subklassen erben Methoden und Attribute der Superklassen; Methoden der Superklassen können überschrieben werden.
- Der Aufruf von Methoden erfolgt durch dynamische Bindung.