

# Informatik I: Einführung in die Programmierung

## 12. Objekt-orientierte Programmierung

Albert-Ludwigs-Universität Freiburg



UNI  
FREIBURG

Prof. Dr. Peter Thiemann

3. Dezember 2025



# Motivation

## Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Was ist Objekt-orientierte Programmierung (OOP)?



UNI  
FREIBURG

- OOP ist ein **Programmierparadigma** (Programmierstil).

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Was ist Objekt-orientierte Programmierung (OOP)?



UNI  
FREIBURG

- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.
  - Zustand global in Datenstrukturen manifestiert.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.
  - Zustand global in Datenstrukturen manifestiert.
  - Funktionen operieren direkt auf dem Zustand.

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.
  - Zustand global in Datenstrukturen manifestiert.
  - Funktionen operieren direkt auf dem Zustand.
- **Objektorientierung**

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.
  - Zustand global in Datenstrukturen manifestiert.
  - Funktionen operieren direkt auf dem Zustand.
- **Objektorientierung**
  - Beschreibung eines Problems anhand kooperierender Objekte.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.
  - Zustand global in Datenstrukturen manifestiert.
  - Funktionen operieren direkt auf dem Zustand.
- **Objektorientierung**
  - Beschreibung eines Problems anhand kooperierender Objekte.
  - Zustand des Programms fragmentiert in den Objekten gespeichert.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Was ist Objekt-orientierte Programmierung (OOP)?



- OOP ist ein **Programmierparadigma** (Programmierstil).
- Eine Art und Weise an ein Problem zu modellieren und zu programmieren.
- Bisher: **Prozedurale Programmierung**
  - Zerlegung des Problems in Datenstrukturen und Funktionen.
  - Zustand global in Datenstrukturen manifestiert.
  - Funktionen operieren direkt auf dem Zustand.
- **Objektorientierung**
  - Beschreibung eines Problems anhand kooperierender Objekte.
  - Zustand des Programms fragmentiert in den Objekten gespeichert.
  - Objekt = Zustand + Operationen darauf.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte (im OOP-Sinne)



UNI  
FREIBURG

- Objekte gibt es im realen Leben überall!

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Objekte (im OOP-Sinne)



UNI  
FREIBURG

- Objekte gibt es im realen Leben überall!
- Objekte haben

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Objekte (im OOP-Sinne)



UNI  
FREIBURG

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

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte (im OOP-Sinne)



- Objekte gibt es im realen Leben überall!
- Objekte haben
  - in der realen Welt: **Zustand** und **Verhalten**
  - in OOP modelliert durch: **Attributwerte** und **Methoden**

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

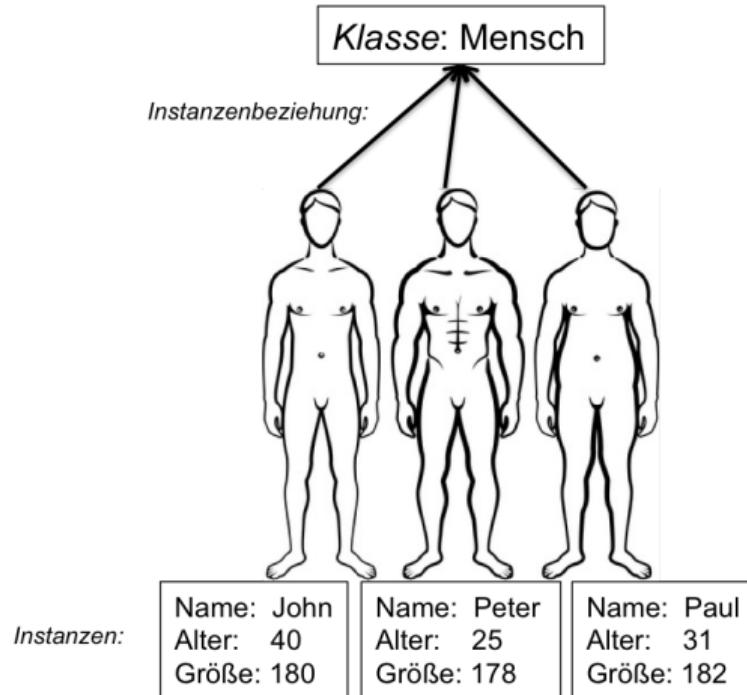
Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte (2)



Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte in OOP

Beispiel: Bankkonto



UNI  
FREIBURG

## Zustand eines Objekts: Attributwerte

Beispiel: Der *Kontostand* wird im Attribut `balance` als Zahl gespeichert.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte in OOP

Beispiel: Bankkonto



UNI  
FREIBURG

## Zustand eines Objekts: Attributwerte

Beispiel: Der *Kontostand* wird im Attribut `balance` als Zahl gespeichert.

## Verhalten eines Objekts: Methoden

Beispiel: Entsprechend einem *Abhebe-Vorgang* verringert ein Aufruf der Methode `withdraw` den Betrag, der unter dem Attribut `balance` gespeichert ist.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte in OOP

Beispiel: Bankkonto



UNI  
FREIBURG

## Zustand eines Objekts: Attributwerte

Beispiel: Der *Kontostand* wird im Attribut `balance` als Zahl gespeichert.

## Verhalten eines Objekts: Methoden

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.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Objekte in OOP

Beispiel: Bankkonto



UNI  
FREIBURG

## Zustand eines Objekts: Attributwerte

Beispiel: Der *Kontostand* wird im Attribut `balance` als Zahl gespeichert.

## Verhalten eines Objekts: Methoden

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.
- Der interne Zustand wird versteckt (**Datenkapselung**).

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Klassen und Objekte (1)



UNI  
FREIBURG

## ■ Eine Klasse

Motivation

Was ist OOP?

Welche Konzepte  
sind wichtig?

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Klassen und Objekte (1)



UNI  
FREIBURG

## ■ Eine Klasse

- ist der „Bauplan“ für bestimmte Objekte;

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

## ■ Eine Klasse

- ist der „Bauplan“ für bestimmte Objekte;
- definiert Attribute und Methoden.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Klassen und Objekte (1)



UNI  
FREIBURG

## ■ Eine Klasse

- ist der „Bauplan“ für bestimmte Objekte;
- definiert Attribute und Methoden.

## ■ Ein Objekt / Instanz der Klasse

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Klassen und Objekte (1)



## ■ Eine Klasse

- ist der „Bauplan“ für bestimmte Objekte;
- definiert Attribute und Methoden.

## ■ Ein Objekt / Instanz der Klasse

- wird dem „Bauplan“ entsprechend erzeugt;

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

## ■ Eine Klasse

- ist der „Bauplan“ für bestimmte Objekte;
- definiert Attribute und Methoden.

## ■ Ein Objekt / Instanz der Klasse

- wird dem „Bauplan“ entsprechend erzeugt;
- Instanzierung sorgt für Initialisierung der Attribute.

Motivation

Was ist OOP?

Welche Konzepte sind wichtig?

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung



# OOP: Die nächsten Schritte

# Wiederholung: Definieren von Klassen

Beispiel: Geometrische Objekte



UNI  
FREIBURG

## Kreis

Ein Kreis wird beschrieben durch seinen Mittelpunkt und seinen Radius.

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Wiederholung: Definieren von Klassen

Beispiel: Geometrische Objekte



UNI  
FREIBURG

## Kreis

Ein Kreis wird beschrieben durch seinen Mittelpunkt und seinen Radius.

## Klassengerüst

```
@dataclass
class Circle:
    x : float
    y : float
    radius : float
```

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Methoden-Definition

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

```
@dataclass
class Circle:
    x : float
    y : float
    radius : float

    def area(self : 'Circle') -> float:
        return (self.radius * self.radius * math.pi)
```

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Methoden-Definition

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

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

```
@dataclass
class Circle:
    x : float
    y : float
    radius : float

    def area(self : 'Circle') -> float:
        return (self.radius * self.radius * math.pi)
```

- Der erste Parameter einer Methode ist speziell und heißt per Konvention **self**.

# Methoden-Definition



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

```
@dataclass
class Circle:
    x : float
    y : float
    radius : float

    def area(self : 'Circle') -> float:
        return (self.radius * self.radius * math.pi)
```

- Der erste Parameter einer Methode ist speziell und heißt per Konvention **self**.
- Dort wird automatisch der Empfänger des Methodenaufrufs übergeben, d.h. die Instanz, auf der die Methode aufgerufen wird.

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition

Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Methoden-Aufrufe



UNI  
FREIBURG

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

- Erzeugung von Instanzen wie gehabt

```
c = Circle(0, 0, 1)
```

- Erzeugung von Instanzen wie gehabt

```
c = Circle(0, 0, 1)
```

- Ein Methodenaufruf geschieht über eine Instanz, die implizit als erstes Argument übergeben wird (`self`-Argument weglassen):

```
print (c.area())
```

liefert die Ausgabe 3.141592653589793 .

# Ein Kreis ...



```
@dataclass
class Circle:
    x : float = 0
    y : float = 0
    radius : float = 1

    def area(self) -> float:
        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
```

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden  
Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Kreise bearbeiten



UNI  
FREIBURG

Objekte sind veränderlich (mutable)

```
c = Circle(x=1, y=2, radius=5)  
print(c.area())
```

Ausgabe: 78.53981633974483

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Kreise bearbeiten



Objekte sind veränderlich (mutable)

```
c = Circle(x=1, y=2, radius=5)
print(c.area())
```

Ausgabe: 78.53981633974483

```
c.size_change(50)
print(c.area())
```

Ausgabe: 19.634954084936208

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Kreise bearbeiten



Objekte sind veränderlich (mutable)

```
c = Circle(x=1, y=2, radius=5)  
print(c.area())
```

Ausgabe: 78.53981633974483

```
c.size_change(50)  
print(c.area())
```

Ausgabe: 19.634954084936208

```
c.move(10, 20)  
print((c.x, c.y))
```

Ausgabe: (11, 22)

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Weitere geometrische Figuren



UNI  
FREIBURG

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden  
Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

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

# Weitere geometrische Figuren



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

# Klasse fürs Rechteck



```
@dataclass
class Rectangle:
    x : float = 0
    y : float = 0
    width : float = 1
    height : float = 1

    def area(self) -> float:
        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
```

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden  
Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Beobachtung



UNI  
FREIBURG

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

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden

Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Beobachtung



- Die Bearbeitung des Referenzpunkts  $(x,y)$  ist bei Circle und Rectangle Objekten gleich.
  - Bei der Konstruktion werden sie gleich behandelt.

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden  
Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Beobachtung



- Die Bearbeitung des Referenzpunkts (x,y) ist bei Circle und Rectangle Objekten gleich.
  - Bei der Konstruktion werden sie gleich behandelt.
  - Die move Methode behandelt sie gleich.

Motivation

OOP: Die  
nächsten  
Schritte

Klassendefinition  
Methoden  
Ein Beispiel: Der  
Kreis

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

- Die Bearbeitung des Referenzpunkts (x,y) ist bei Circle und Rectangle Objekten gleich.
  - Bei der Konstruktion werden sie gleich behandelt.
  - Die move Methode behandelt sie gleich.
- OOP liefert eine **Abstraktion**, mit der diese Gemeinsamkeit ausgedrückt werden kann, sodass die Spezifikation der Attribute und die move Methode nur einmal geschrieben werden müssen.

# Vererbung

Motivation

OOP: Die  
nächsten  
Schritte

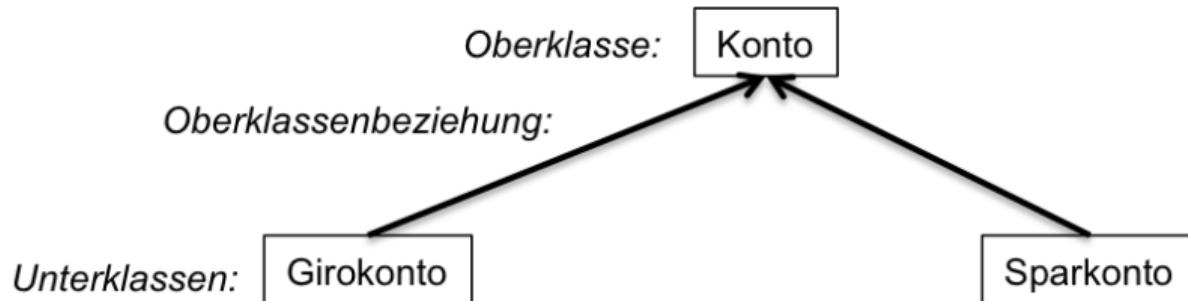
Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

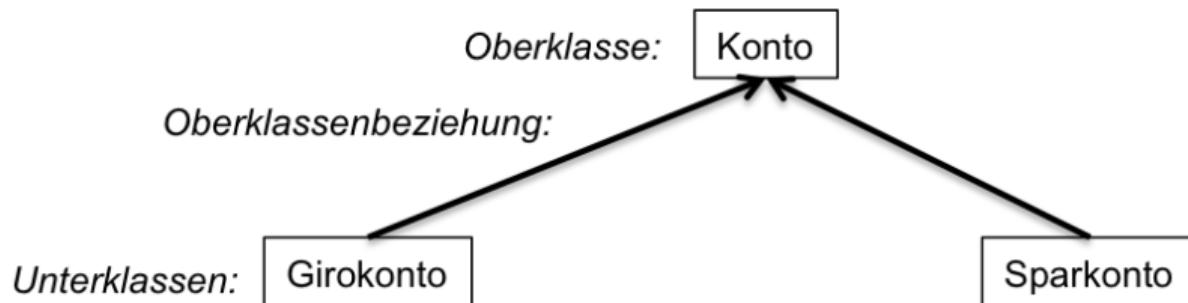
Zusammen-  
fassung

# Klassenhierarchien



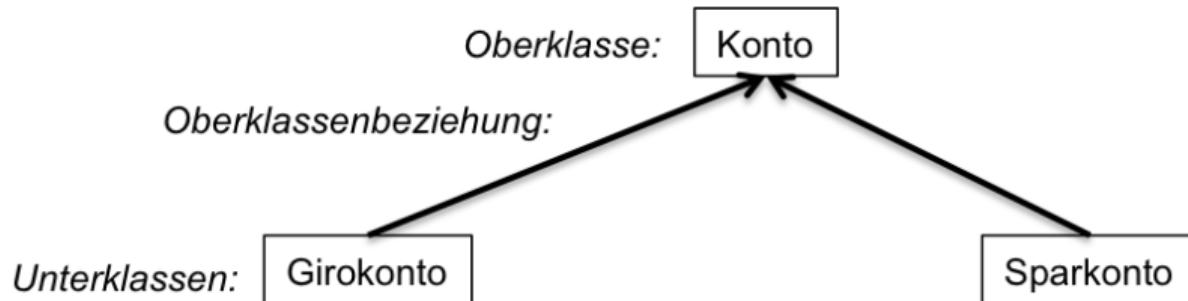
- Klassen können in einer **Vererbungshierarchie** angeordnet werden:  
Girokonto und Sparkonto **erben** von Konto

# Klassenhierarchien



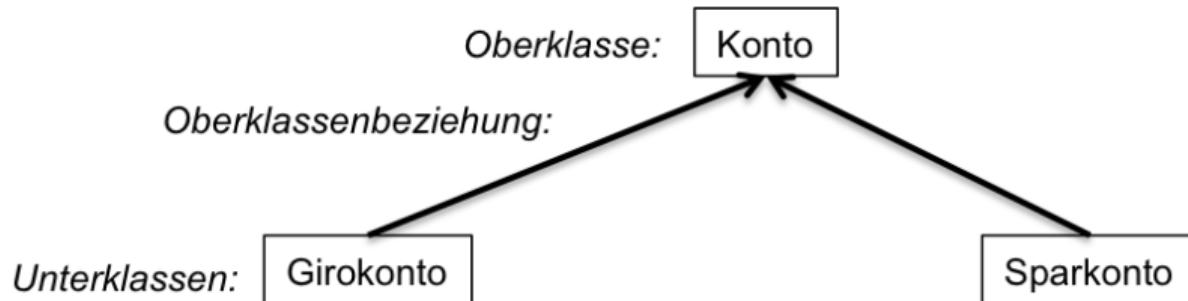
- Klassen können in einer **Vererbungshierarchie** angeordnet werden:  
Girokonto und Sparkonto **erben** von Konto
- Die oberen Klassen sind allgemeiner, die unteren spezieller.

# Klassenhierarchien



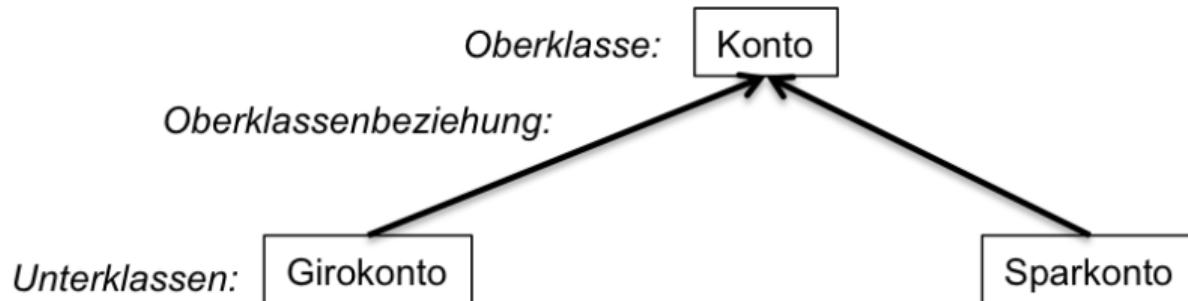
- Klassen können in einer **Vererbungshierarchie** angeordnet werden:  
Girokonto und Sparkonto **erben** von Konto
- Die oberen Klassen sind allgemeiner, die unteren spezieller.
- Terminologie:

# Klassenhierarchien



- Klassen können in einer **Vererbungshierarchie** angeordnet werden:  
Girokonto und Sparkonto **erben** von Konto
- Die oberen Klassen sind allgemeiner, die unteren spezieller.
- Terminologie:
  - Superklasse, Oberklasse oder Basisklasse (für die obere Klasse)

# Klassenhierarchien



- Klassen können in einer **Vererbungshierarchie** angeordnet werden:  
Girokonto und Sparkonto **erben** von Konto
- Die oberen Klassen sind allgemeiner, die unteren spezieller.
- Terminologie:
  - Superklasse, Oberklasse oder Basisklasse (für die obere Klasse)
  - Subklasse, Unterklasse oder abgeleitete Klasse (für die unteren Klassen)

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

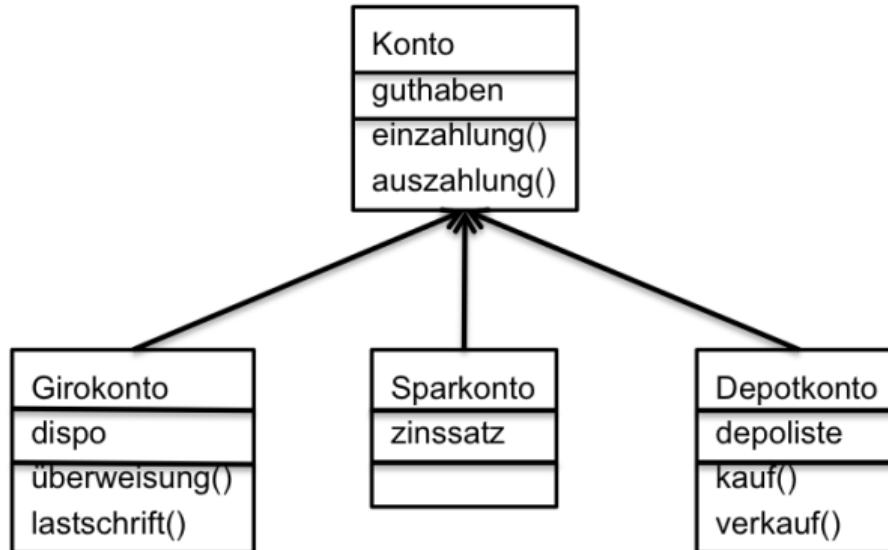
Ein bisschen  
GUI

Zusammen-  
fassung

# Vererbung



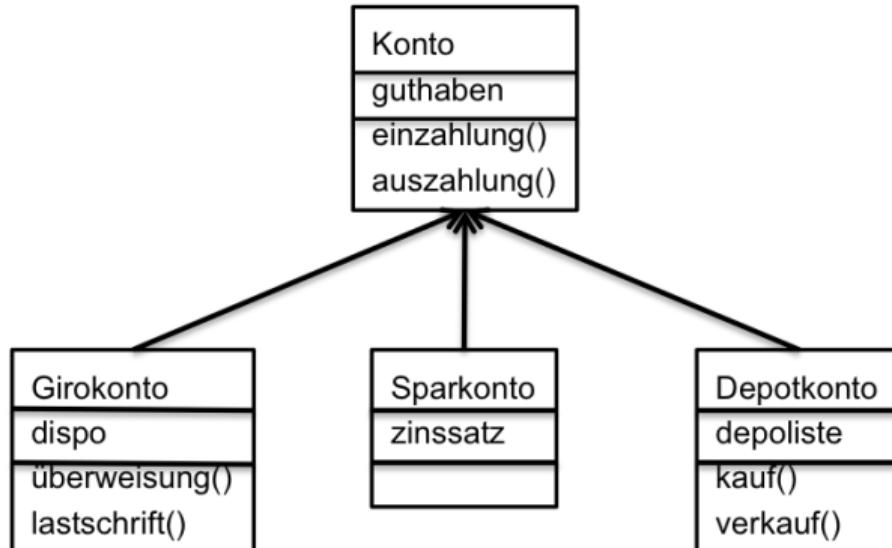
- Unterklassen **erben** Attribute und Methoden von der Oberklasse.



# Vererbung

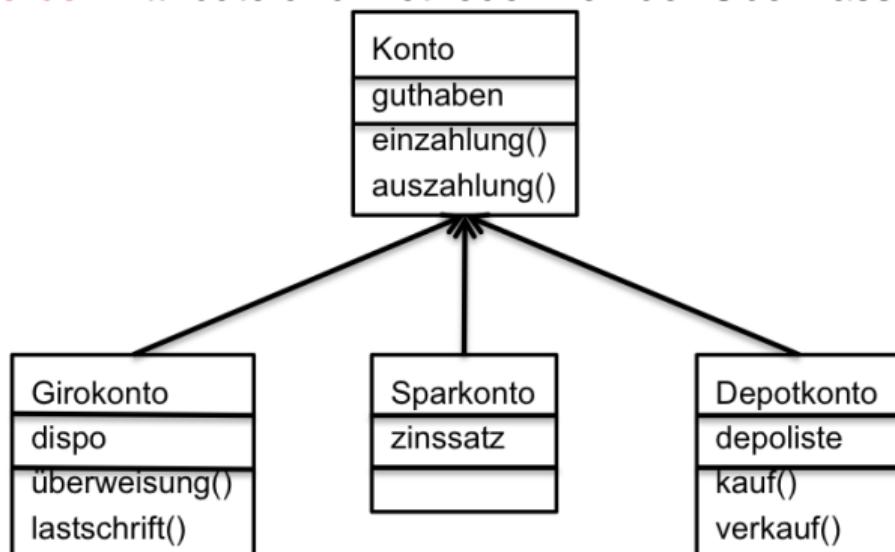


- Unterklassen **erben** Attribute und Methoden von der Oberklasse.



- ...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**.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

**Vererbung  
konkret**

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Vererbung konkret

# Weitere Objekte



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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

**Vererbung  
konkret**

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Weitere Objekte



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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

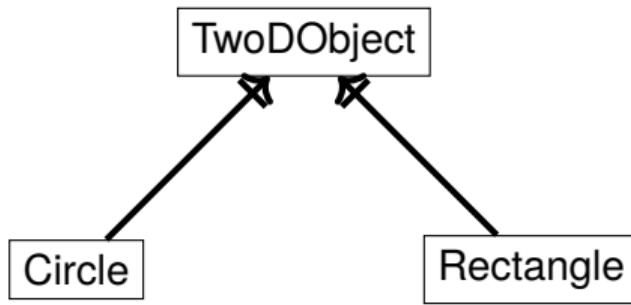
Ein bisschen  
GUI

Zusammen-  
fassung

# Weitere Objekte



- Fasse **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**:



Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

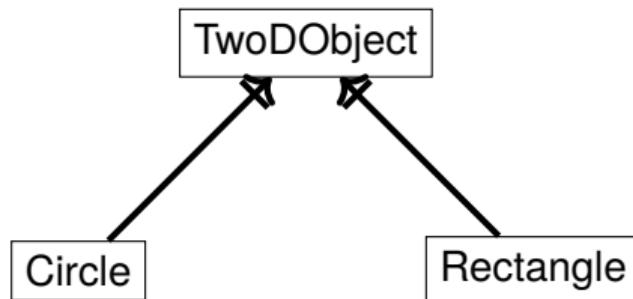
Ein bisschen  
GUI

Zusammen-  
fassung

# Weitere Objekte



- Fasse **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**.

# 2D-Objekte



- Allen geometrischen Figuren ist gemeinsam, dass sie einen Referenzpunkt besitzen, der verschoben werden kann, und dass sie eine Fläche besitzen.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# 2D-Objekte



- Allen geometrischen Figuren ist gemeinsam, dass sie einen Referenzpunkt besitzen, der verschoben werden kann, und dass sie eine Fläche besitzen.

geoclasses.py (1)

```
@dataclass
class TwoDObject:
    x : float = 0
    y : float = 0

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

    def area(self) -> float:
        return 0
```

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Ein Kreis ist ein 2D-Objekt



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

geoclasses.py (2)

```
@dataclass
class Circle(TwoDObject):
    radius : float = 1

    def area(self) -> float:
        return self.radius * self.radius * 3.14

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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)



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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)



- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)



- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`
- Die geerbte Methode `move` wird übernommen.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)



- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
- Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`
- Die geerbte Methode `move` wird übernommen.
- Die geerbte Methode `area` wird überschrieben, weil wir in der Subklasse eine neue Definition angeben.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)

- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
  - Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`
  - Die geerbte Methode `move` wird übernommen.
  - Die geerbte Methode `area` wird überschrieben, weil wir in der Subklasse eine neue Definition angeben.
- ⇒ Jede geerbte Methode wird entweder übernommen oder überschrieben!

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)



- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
  - Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`
  - Die geerbte Methode `move` wird übernommen.
  - Die geerbte Methode `area` wird überschrieben, weil wir in der Subklasse eine neue Definition angeben.
- ⇒ Jede geerbte Methode wird entweder übernommen oder überschrieben!
- Auf einer `Circle` Instanz wird aufgerufen

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Überschreiben (*override*)

- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
  - Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`
  - Die geerbte Methode `move` wird übernommen.
  - Die geerbte Methode `area` wird überschrieben, weil wir in der Subklasse eine neue Definition angeben.
- ⇒ Jede geerbte Methode wird entweder übernommen oder überschrieben!
- Auf einer `Circle` Instanz wird aufgerufen
    - `move` aus `TwoDObject`

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte  
Überschreiben und dynamische Bindung

Ein bisschen GUI

Zusammenfassung

# Überschreiben (*override*)

- Durch Vererbung kommen weitere Attribute und Methoden hinzu (hier: `move` und `area` werden von der Superklasse `TwoDObject` geerbt).
  - Die neuen Attribute werden in der Argumentliste des Konstruktors hinten angehängt. Beispiel: `Circle(x, y, radius)`
  - Die geerbte Methode `move` wird übernommen.
  - Die geerbte Methode `area` wird überschrieben, weil wir in der Subklasse eine neue Definition angeben.
- ⇒ Jede geerbte Methode wird entweder übernommen oder überschrieben!
- Auf einer `Circle` Instanz wird aufgerufen
    - `move` aus `TwoDObject`
    - `area` aus `Circle`

Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

2D-Objekte  
Überschreiben und dynamische Bindung

Ein bisschen GUI

Zusammenfassung

# Dynamische Bindung (Dynamic Dispatch)



- Das Verhalten eines Methodenaufrufs wie `obj.area()` oder `obj.move()` wird erst zur Laufzeit des Programms bestimmt.
- Es hängt vom (Laufzeit-) Typ von `obj` ab.
  - Falls `type(obj) == TwoDObject`, dann wird sowohl für `area` als auch für `move` der Code aus `TwoDObject` verwendet.
  - 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** oder **dynamic dispatch** und ist charakteristisch für objekt-orientierte Sprachen.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung



# 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)
```

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret2D-Objekte  
Überschreiben und  
dynamische  
BindungEin bisschen  
GUIZusammen-  
fassung

# Kreissektoren



- Ein Kreissektor wird beschrieben durch einen Kreis, einen Startwinkel und einen Endwinkel:

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

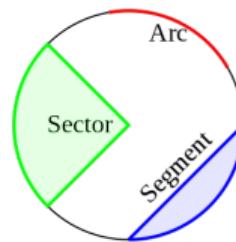
Ein bisschen  
GUI

Zusammen-  
fassung

# Kreissektoren



- Ein Kreissektor wird beschrieben durch einen Kreis, einen Startwinkel und einen Endwinkel:



[https://commons.wikimedia.org/wiki/File:Circle\\_slices.svg](https://commons.wikimedia.org/wiki/File:Circle_slices.svg) (public domain)

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

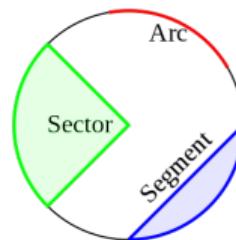
Ein bisschen  
GUI

Zusammen-  
fassung

# Kreissektoren



- Ein Kreissektor wird beschrieben durch einen Kreis, einen Startwinkel und einen Endwinkel:



[https://commons.wikimedia.org/wiki/File:Circle\\_slices.svg](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



```
@dataclass
class Sector (Circle):
    start_angle : float = 0
    end_angle : float = 180

    def area(self) -> float:
        circle_fraction = (self.end_angle - self.start_angle) / 360
        return self.radius * self.radius * math.pi * circle_fraction
```

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Kreissektor als Subklasse vom Kreis



```
@dataclass
class Sector (Circle):
    start_angle : float = 0
    end_angle : float = 180

    def area(self) -> float:
        circle_fraction = (self.end_angle - self.start_angle) / 360
        return self.radius * self.radius * math.pi * circle_fraction
```

Eine Instanz von Sector verwendet ...

- move von TwoDObject
- size\_change von Circle
- area von Sector, aber ein Teil des Codes ist aus Circle kopiert!

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Super!



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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Super!



- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Über `super()` kann die überschriebene Methode in einer Superklasse aufgerufen werden.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Super!



- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Über `super()` kann die überschriebene Methode in einer Superklasse aufgerufen werden.

## Verwendung von `super`

```
@dataclass
class Sector(Circle): ...
    def area(self) -> float:
        circle_fraction = (self.end_angle - self.start_angle) / 360
        return super().area() * circle_fraction
```

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Über `super()` kann die überschriebene Methode in einer Superklasse aufgerufen werden.

## Verwendung von `super`

```
@dataclass
class Sector(Circle): ...
    def area(self) -> float:
        circle_fraction = (self.end_angle - self.start_angle) / 360
        return super().area() * circle_fraction
```

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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Über `super()` kann die überschriebene Methode in einer Superklasse aufgerufen werden.

## Verwendung von `super`

```
@dataclass
class Sector(Circle): ...
    def area(self) -> float:
        circle_fraction = (self.end_angle - self.start_angle) / 360
        return super().area() * circle_fraction
```

- `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.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

- Was, wenn die `area()` Methode in der Subklasse `Sector` eine Methode aus der Superklasse `Circle` verwenden könnte?
- Über `super()` kann die überschriebene Methode in einer Superklasse aufgerufen werden.

## Verwendung von `super`

```
@dataclass
class Sector(Circle): ...
    def area(self) -> float:
        circle_fraction = (self.end_angle - self.start_angle) / 360
        return super().area() * circle_fraction
```

- `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.
- D.h. Von `area` in `Sector` wird `area` in `Circle` aufgerufen.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Beispiel



```
s = Sector (x=1, y=2, radius=5, end_angle=90)
print(s.area())
```

Ausgabe: 19.634954084936208

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung



# Beispiel

```
s = Sector (x=1, y=2, radius=5, end_angle=90)
print(s.area())
```

Ausgabe: 19.634954084936208

```
c = Circle (x=1, y=2, radius=5)
print(c.area())
```

Ausgabe: 78.5

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung



# Beispiel

```
s = Sector (x=1, y=2, radius=5, end_angle=90)
print(s.area())
```

Ausgabe: 19.634954084936208

```
c = Circle (x=1, y=2, radius=5)
print(c.area())
```

Ausgabe: 78.5

```
assert math.isclose(s.area() * 4, c.area(), rel_tol=0.01)
s.move(9,8)
print((s.x, s.y))
```

Ausgabe: (10, 10)

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung



# Beispiel

```
s = Sector (x=1, y=2, radius=5, end_angle=90)
print(s.area())
```

Ausgabe: 19.634954084936208

```
c = Circle (x=1, y=2, radius=5)
print(c.area())
```

Ausgabe: 78.5

```
assert math.isclose(s.area() * 4, c.area(), rel_tol=0.01)
s.move(9,8)
print((s.x, s.y))
```

Ausgabe: (10, 10)

```
s.size_change(200)
print(s.area())
```

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Ein Rechteck ist auch ein 2D-Objekt



- Und weiter geht es mit Rechtecken

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Ein Rechteck ist auch ein 2D-Objekt



## ■ Und weiter geht es mit Rechtecken

geoclasses.py (5)

```
@dataclass
class Rectangle(TwoDObject):
    height : float = 1
    width : float = 1

    def area(self) -> float:
        return self.height * self.width

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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Spielen mit 2D-Objekten



```
t = TwoDObject(x=10, y=20)
c = Circle(5,11,22)
r = Rectangle(100,100,20,20)
print ((c.x, c.y)); c.move (89,78); print ((c.x, c.y))
```

Ausgabe: (5, 11) (94, 89)

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Spielen mit 2D-Objekten



```
t = TwoDObject(x=10, y=20)
c = Circle(5,11,22)
r = Rectangle(100,100,20,20)
print ((c.x, c.y)); c.move (89,78); print ((c.x, c.y))
```

Ausgabe: (5, 11) (94, 89)

```
print (f"t.area= {t.area()}, r.area= {r.area()}")
```

Ausgabe: t.area= 0, r.area= 400

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Spielen mit 2D-Objekten



```
t = TwoDObject(x=10, y=20)
c = Circle(5,11,22)
r = Rectangle(100,100,20,20)
print ((c.x, c.y)); c.move (89,78); print ((c.x, c.y))
```

Ausgabe: (5, 11) (94, 89)

```
print (f"t.area= {t.area()}, r.area= {r.area()}")
```

Ausgabe: t.area= 0, r.area= 400

```
r.size_change(50); print(r.area())
```

Ausgabe: 100.0

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

2D-Objekte  
Überschreiben und  
dynamische  
Bindung

Ein bisschen  
GUI

Zusammen-  
fassung

# Ein bisschen GUI

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Graphical User Interface



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

# Graphical User Interface



- Jede moderne Programmiersprache bietet heute APIs (**Application Programming Interface**) für GUIs (**Graphical User Interface**) an.  
⇒ Möglichkeit, interaktiv per Fenster, Tastatur und Maus Ein- und Ausgaben zu einem Programm zu bearbeiten.

# Graphical User Interface



- Jede moderne Programmiersprache bietet heute APIs (**Application Programming Interface**) für GUIs (**Graphical User Interface**) an.
  - ⇒ Möglichkeit, interaktiv per Fenster, Tastatur und Maus Ein- und Ausgaben zu einem Programm zu bearbeiten.
- Für Python gibt es **tkinter** (integriert), **PyGtk**, **wxWidget**, **PyQt**, uvam.

# Graphical User Interface



- Jede moderne Programmiersprache bietet heute APIs (**Application Programming Interface**) für GUIs (**Graphical User Interface**) an.
  - ⇒ Möglichkeit, interaktiv per Fenster, Tastatur und Maus Ein- und Ausgaben zu einem Programm zu bearbeiten.
- Für Python gibt es **tkinter** (integriert), **PyGtk**, **wxWidget**, **PyQt**, uvam.
- Wir wollen jetzt einen kleinen Teil von **tkinter** kennenlernen, um unsere Geo-Objekte zu visualisieren.

# Hello World



```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte intern durch einen Baum.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Hello World



```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte intern durch einen Baum.
- `root` wird das Wurzelobjekt, in das alle anderen Objekte eingehängt werden.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Hello World



```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte intern durch einen Baum.
- `root` wird das Wurzelobjekt, in das alle anderen Objekte eingehängt werden.
- `lab` repräsentiert ein **Label-Widget** innerhalb des `root`-Objekts.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Hello World



```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte intern durch einen Baum.
- `root` wird das Wurzelobjekt, in das alle anderen Objekte eingehängt werden.
- `lab` repräsentiert ein **Label-Widget** innerhalb des `root`-Objekts.
  - Ein **Widget** ist eine (meist rechteckige) Fläche auf dem Schirm, auf der eine bestimmte Ein-/Ausgabefunktionalität implementiert ist.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Hello World



```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte intern durch einen Baum.
- `root` wird das Wurzelobjekt, in das alle anderen Objekte eingehängt werden.
- `lab` repräsentiert ein **Label-Widget** innerhalb des `root`-Objekts.
  - Ein **Widget** ist eine (meist rechteckige) Fläche auf dem Schirm, auf der eine bestimmte Ein-/Ausgabefunktionalität implementiert ist.
  - Das Label-Widget zeigt einen String als Text an. Es verarbeitet keine Eingaben.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Hello World



```
import tkinter as tk

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

- `tkinter` repräsentiert Bildschirminhalte intern durch einen Baum.
- `root` wird das Wurzelobjekt, in das alle anderen Objekte eingehängt werden.
- `lab` repräsentiert ein **Label-Widget** innerhalb des `root`-Objekts.
  - Ein **Widget** ist eine (meist rechteckige) Fläche auf dem Schirm, auf der eine bestimmte Ein-/Ausgabefunktionalität implementiert ist.
  - Das Label-Widget zeigt einen String als Text an. Es verarbeitet keine Eingaben.
- Mit `lab.pack()` wird das Widget `lab` in seinem Elternfenster positioniert.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Objekte malen



```
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 Zeichenfläche (Leinwand) funktioniert, auf der geometrische Figuren gemalt werden können.
- Der Konstruktor für `tk.Canvas` nimmt Höhe und Breite in **Pixeln** (Bildpunkten).

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung



Hello World



Motivation

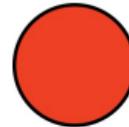
OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

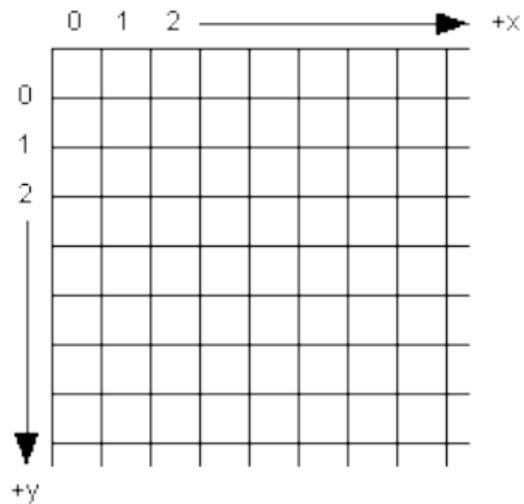
Zusammen-  
fassung



# Grafik-Koordinatensysteme



- Die Positionierung auf dem Canvas erfolgt über ein Koordinatensystem.



Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

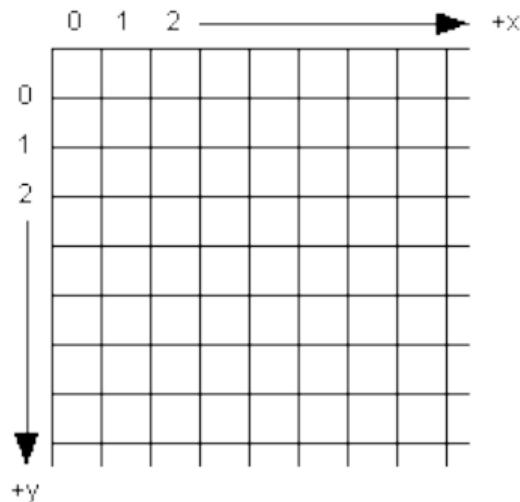
Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Grafik-Koordinatensysteme

- Die Positionierung auf dem Canvas erfolgt über ein Koordinatensystem.
- Im Unterschied zum mathematischen Koordinatensystem liegt der Nullpunkt bei Grafikdarstellungen immer **oben links**.



Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

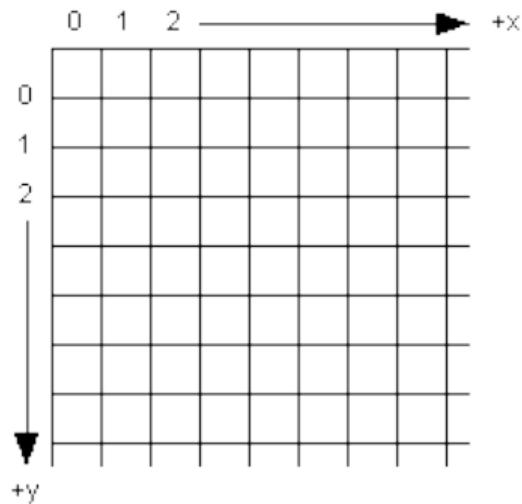
Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Grafik-Koordinatensysteme

- Die Positionierung auf dem Canvas erfolgt über ein Koordinatensystem.
- Im Unterschied zum mathematischen Koordinatensystem liegt der Nullpunkt bei Grafikdarstellungen immer **oben links**.
- Wie gewohnt dienen **(x,y)**-Paare zur Bestimmung von Punkten.



Motivation

OOP: Die nächsten Schritte

Vererbung

Vererbung konkret

Ein bisschen GUI

Zusammenfassung

# Einige Canvas-Methoden



- `canvas.create_line(x1, y1, x2, y2, **options)`  
**Linie** von  $(x_1, y_1)$  nach  $(x_2, y_2)$ .

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Einige Canvas-Methoden

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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Einige Canvas-Methoden



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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Einige Canvas-Methoden



- `canvas.create_line(x1, y1, x2, y2, **options)`  
**Linie** von (x1, y1) nach (x2, y2).
- `canvas.create_rectangle(x1, y1, x2, y2, **options)`  
**Rechteck** mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- `canvas.create_oval(x1, y1, x2, y2, **options)`  
**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, eine eindeutige Zahl, mit der das Objekt manipuliert werden kann.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Einige Canvas-Methoden



- `canvas.create_line(x1, y1, x2, y2, **options)`  
**Linie** von (x1, y1) nach (x2, y2).
- `canvas.create_rectangle(x1, y1, x2, y2, **options)`  
**Rechteck** mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- `canvas.create_oval(x1, y1, x2, y2, **options)`  
**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, eine eindeutige Zahl, mit der das Objekt manipuliert werden kann.
- `canvas.delete(i)` **löscht** Objekt mit dem Index *i*.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Einige Canvas-Methoden



- `canvas.create_line(x1, y1, x2, y2, **options)`  
**Linie** von (x1, y1) nach (x2, y2).
- `canvas.create_rectangle(x1, y1, x2, y2, **options)`  
**Rechteck** mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- `canvas.create_oval(x1, y1, x2, y2, **options)`  
**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, eine eindeutige Zahl, mit der das Objekt manipuliert werden kann.
- `canvas.delete(i)` **löscht** Objekt mit dem Index *i*.
- `canvas.move(i, xdelta, ydelta)` **bewegt** Objekt *i* um xdelta und ydelta.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Einige Canvas-Methoden



- `canvas.create_line(x1, y1, x2, y2, **options)`  
**Linie** von (x1, y1) nach (x2, y2).
- `canvas.create_rectangle(x1, y1, x2, y2, **options)`  
**Rechteck** mit oberer linker Ecke (x1, y1) und unterer rechter Ecke (x2, y2).
- `canvas.create_oval(x1, y1, x2, y2, **options)`  
**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, eine eindeutige Zahl, mit der das Objekt manipuliert werden kann.
- `canvas.delete(i)` **löscht** 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



## Geoclasses visuell

```
from dataclasses import dataclass, field
@dataclass
class TwoDObjectV:
    cv : tk.Canvas
    x : float = 0
    y : float = 0
    index : int = field(default= 0, init= False)

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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

- `field(default= 0, init= False)` dieses Attribut hat Standardwert 0, wird aber nicht vom Konstruktor gesetzt.

# Visualisierung



## Geoclasses visuell

```
@dataclass
class CircleV(TwoDObjectV):
    radius : float = 1

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

## Hintergrund des Entwurfs

- Der Aufruf des Konstruktors erzeugt das Objekt auf dem Canvas.  
`CircleV (canvas, 10, 10)`
- Die Methoden wirken gleichzeitig auf das Canvas-Objekt.

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung

# Zusammenfassung

# Zusammenfassung



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

Motivation

OOP: Die  
nächsten  
Schritte

Vererbung

Vererbung  
konkret

Ein bisschen  
GUI

Zusammen-  
fassung