CloudCommand

# Cyber Security

CloudCommand GmbH chr.schumacher@gmx.tm

CloudCommand

# Grundlagen der Programmierung

CloudCommand GmbH chr.schumacher@gmx.tm

# Einführung in OOP

# Grundlagen

**Einführung in Programmierparadigmen**

In der Welt der Softwareentwicklung gibt es verschiedene Ansätze, um Probleme zu lösen und Programme zu strukturieren.

Zwei der grundlegenden Paradigmen sind:

- die imperative Programmierung
- die objektorientierte Programmierung

# Grundlagen

**Was ist imperative Programmierung?**
Der Begriff **imperativ** stammt im weiteren Sinne vom lateinischen **imperare** ab, was soviel bedeutet wie **befehlen**.
Imperative Programmierung ist also ein Paradigma welches dem Computer genau befiehlt wie es eine Aufgabe erledigen soll. Die Anweisungen bestehen aus einer geordneten Abfolge.

- Befehlsorientierter Ansatz
- Sequenzielle Ausführung von Anweisungen
- Beispiele für imperative Sprachen: C, Java, Python

# Grundlagen

**Vorteile der imperativen Programmierung**

Die imperative Programmierung bietet bestimmte Vorteile, die sie für viele Anwendungsfälle attraktiv macht.

- Einfachheit und Klarheit bei kleinen Programmen
- Direkte Kontrolle über die Ausführung

# Grundlagen

**Was ist objektorientierte Programmierung (OOP)?**
Die objektorientierte Programmierung ist ein Paradigma, bei dem das Softwaredesign auf Daten oder Objekten basiert. OOP fördert eine stärkere Abstraktion und Modularität.

- Datenkapselung und -abstraktion
- Vererbung (und Polymorphismus)
- Beispiele für OOP-Sprachen: C++, Java, Python

# Grundlagen

**Vorteile der objektorientierten Programmierung**

Die objektorientierte Programmierung bietet Vorteile, die sie für die Entwicklung großer und komplexer Systeme geeignet machen.

- Erleichterte Wartung und Erweiterung von Code
- Natürliche Modellierung realer Entitäten
- Wiederverwendbarkeit durch Vererbung

# Grundlagen

**Herausforderungen der objektorientierten Programmierung**
Auch die OOP hat ihre Herausforderungen, die Entwickler im Auge behalten müssen.

- Übermäßige Komplexität durch tiefe Vererbungshierarchien
- Mögliche Performanceeinbußen durch Abstraktion
- Komplexere Parallelisierung von OO-Programmen

# Konzepte

**Objekte und Klassen in der OOP**

- **Klassen** sind benutzerdefinierte Datentypen, die als Blaupause für individuelle Objekte, Attribute und Methoden dienen.

- **Objekte** sind Instanzen einer Klasse, die mit speziell definierten Daten erstellt werden. Sie können realen Objekten oder abstrakten Konzepten entsprechen.

# Konzepte

**Attribute und Methoden in der OOP**

- **Attribute** sind Eigenschaften eines Objektes. Sie repräsentieren den Zustand eines Objektes und werden in den zugehörigen Variablen gespeichert.

- **Methoden** sind Funktionen die auf das Objekt angewendet werden. Sie beschreiben das Verhalten eines Objektes.

# Konzepte

**Der “self” operator in Python**

- Das **self**-Schlüsselwort in Python ist ein Verweis auf das aktuelle Objekt, auf das eine Methode angewendet wird.
- Es wird automatisch als erster Parameter an jede Methode einer Klasse übergeben, wenn sie aufgerufen wird.

# Konzepte

**Konstruktoren und Destruktoren in der OOP**

- **Konstruktoren** sind spezielle Methoden, die beim Erzeugen von Objekten aufgerufen werden. Ihre Aufgabe ist es, Objekte in einen definierten Anfangszustand zu versetzen.

  **def __init__(self):**

- **Destruktoren** werden beim Auflösen von Objekten aufgerufen. Sie dienen dazu, Ressourcen freizugeben und Aufräumarbeiten durchzuführen.

  **def __del__(self):**

# Konzepte

- Der **self** Operator in Python ist ein Verweis auf das aktuelle Objekt
- Die **__init__** Methode in Python ist der Konstruktor welches beim erstellen eines neuen Objektes aufgerufen wird

```python
class meineNummer:
    def __init__(self, wert):
        self.wert = wert

    def print_wert(self):
        print(self.wert)

# Instanz der Klasse meineNummer erstellen
obj1 = meineNummer(17)

# Methode print_wert aufrufen
obj1.print_wert()  # Ausgabe: 17
```

# Konzepte

- **Klassen** sind benutzerdefinierte Datentypen, die als Blaupause für individuelle Objekte, Attribute und Methoden dienen.

```
class Rechteck:
    def __init__(self, breite, höhe):
        self.breite = breite
        self.höhe = höhe

    def fläche(self):
        return self.breite * self.höhe
```

# Konzepte

- **Objekte** sind Instanzen einer Klasse

```python
class Rechteck:
    def __init__(self, breite, höhe):
        self.breite = breite
        self.höhe = höhe

    # Methode
    def fläche(self):
        return self.breite * self.höhe


# Instanz der Klasse Rechteck erstellen
mein_rechteck = Rechteck(5, 10)

# Fläche des Rechtecks berechnen
fläche = mein_rechteck.fläche()
```

# Konzepte

**Vererbung in der OOP**

Vererbung ermöglicht es, aufbauend auf bestehenden Klassen neue Klassen zu erstellen. Die Beziehung zwischen der Ursprünglichen und der neuen Klasse bleibt bestehen.

- Basisklassen und abgeleitete Klassen
- Super- und Subklassen-Beziehungen
- Überschreiben von Methoden

# Konzepte

**Vererbung in der OOP - Basisklassen und abgeleitete Klassen**

- Die bestehende Klasse wird als **Basisklasse** (auch Super-, Ober- oder Elternklasse) bezeichnet.

- Die neue Klasse, die von der Basisklasse erbt, nennt sich abgeleitete Klasse (auch Sub-, Unter- oder **Kindklasse**).

# Konzepte

**Vererbung in der OOP**

- **Basisklasse** in Python
  **class Shape:**

- **abgeleitete Klasse** in Python
  **class Circle(Shape):**

- zugriff in der eigenen Klasse mit dem Keyword **self** möglich
- zugriff auf Basisklasse mit der Funktion **super()** möglich
- methoden der Basisklasse können in der Subklasse überschrieben werden

# Konzepte

```python
# Basisklasse
class Tier:
    def __init__(self):
        self.stamm = "Wirbeltier"

# Abgeleitete Klasse
class Hund(Tier):
    def __init__(self, rasse):
        self.rasse = rasse

# Instanz der abgeleiteten Klasse erstellen
hund = Hund("Shiba Inu")
```

# Konzepte

**Methodenüberschreibung - vererbung**

- Die Methodenüberschreibung ermöglicht es einer abgeleiteten Klasse, eine Methode der Basisklasse mit einer eigenen Implementierung zu überschreiben. Dies ermöglicht eine spezialisierte Implementation von Unterklasse.

- Methodenüberschreibung tritt auf, wenn eine Methode in der abgeleiteten Klasse die gleiche Signatur wie eine Methode in der Basisklasse hat.

# Konzepte

## Methodenüberschreibung - vererbung

```python
class Tier:
    def laut_machen(self):
        return "Ein generisches Geräusch"

class Hund(Tier):
    def laut_machen(self):
        return "Wuff!"

class Katze(Tier):
    def laut_machen(self):
        return "Miau!"

# Instanziierung der Objekte
tier = Tier()
hund = Hund()
katze = Katze()

# Aufruf der Methoden
print(tier.laut_machen())  # Ausgabe: Ein generisches Geräusch
print(hund.laut_machen())  # Ausgabe: Wuff!
print(katze.laut_machen())  # Ausgabe: Miau!
```

# Konzepte

**Elternklassenzugriff - super**

- **super()** wird verwendet, um auf die Methoden und Eigenschaften der Basisklasse zuzugreifen ohne diese zu überschreiben

# Konzepte

## Elternklassenzugriff - super

```python
class Tier:
    def __init__(self, name):
        self.name = name

class Hund(Tier):
    def __init__(self, name, rasse):
        super().__init__(name)
        self.rasse = rasse

class Katze(Tier):
    def __init__(self, name, farbe):
        super().__init__(name)
        self.farbe = farbe

# Instanziierung der Objekte
hund = Hund("Bello", "Schäferhund")
katze = Katze("Minka", "Schwarz")
```

```python
# Zugriff auf die Attribute

print(f"Der Name des Hundes ist: {hund.name}")
# Ausgabe: Der Name des Hundes ist: Bello

print(f"Die Rasse des Hundes ist: {hund.rasse}")
# Ausgabe: Die Rasse des Hundes ist: Schäferhund

print(f"Der Name der Katze ist: {katze.name}")
# Ausgabe: Der Name der Katze ist: Minka

print(f"Die Farbe der Katze ist: {katze.farbe}")
# Ausgabe: Die Farbe der Katze ist: Schwarz
```

# Konzepte

**Zugriffsmodifikatoren in der OOP**

Mitglieder einer Klasse sind mit den folgenden Zugriffsrechten definierbar:

- **public:** sind aus allen Teilen eines Programmes zugreifbar. Alle Mitglieder einer Klasse sind standardmäßig public. In Python ist keine weitere definierung notwendig
- **protected:** sind nur innerhalb der selben oder von Erben dieser Klasse zugreifbar. In Python werden sie durch einen _ vor dem tatsächlichen namen definiert:

  **_meineVariable = “ist PROTECTED”**
- **private:** sind nur innerhalb der selben Klasse zugreifbar. In Python werden sie durch einen **doppelten _** vor dem tatsächlichen namen definiert:

  **__meineVariable2 = “ist PRIVATE”**

# Konzepte

**Zugriffsmethoden in der OOP**

Die Modifikation von Objekt innerhalb der OO-Programmierung läuft in der Regel

- **getter-methode:** In Python können Getter-Methoden verwendet werden, um den Zugriff auf private oder geschützte Attribute zu ermöglichen.
- Typischerweise werden sie mit **get_** gefolgt vom Namen der Variablen benannt.

```
class MeineKlasse:
    def __init__(self, wert):
        self._meinWert = wert  # _meinWert ist protected

    def get_meinWert(self):
        return self._meinWert

obj = MeineKlasse(42)
print(obj.get_meinWert())  # Ausgabe: 42
```

# Konzepte

**Zugriffsmethoden in der OOP**

Die Modifikation von Objekt innerhalb der OO-Programmierung läuft in der Regel

- **setter-methode:** In Python können Setter-Methoden verwendet werden, um den Wert von privaten oder geschützten Attributen zu ändern.
- Typischerweise werden sie mit **set_** gefolgt vom Namen der Variablen benannt.

# Konzepte

## Zugriffsmethoden in der OOP

```python
class MeineKlasse:
    def __init__(self, wert):
        self._meinWert = wert  # _meinWert ist protected

    def set_meinWert(self, wert):
        self._meinWert = wert

    def get_meinWert(self):
        return self._meinWert

obj = MeineKlasse(42)
obj.set_meinWert(100)
print(obj.get_meinWert())  # Ausgabe: 100
```

# Konzepte

**Zugriffsmethoden in der OOP**

Eine alternative für die Modifikation von Objekt innerhalb der OO-Programmierung

- **property-Dekorator:** Diese ermöglichen es, Methoden wie Attribute zu verwenden.
- Die Methode, die mit dem property-Dekorator dekoriert ist, wird als **Getter-Methode** behandelt und gibt den Wert der Eigenschaft zurück.
- Um eine Setter-Methode zu definieren, kann der property-Dekorator speziell eine **Setter-Methode** definieren, die den Wert der Eigenschaft ändert.

# Konzepte

## Zugriffsmethoden in der OOP

```python
class MeineKlasse:
    def __init__(self, wert):
        self._meinWert = wert  # _meinWert ist protected

    @property
    def meinWert(self):
        return self._meinWert

    @meinWert.setter
    def meinWert(self, wert):
        self._meinWert = wert

obj = MeineKlasse(42)
print(obj.meinWert)  # Ausgabe: 42
obj.meinWert = 100
print(obj.meinWert)  # Ausgabe: 100
```

# CloudCommand

CloudCommand GmbH chr.schumacher@gmx.tm