# Klassen und Objekte:

In der objektorientierten Programmierung (OOP) ist eine Klasse eine Vorlage oder ein Bauplan für Objekte. Eine Klasse definiert eine Gruppe von Attributen (Daten) und Methoden (Funktionen), die den Zustand und das Verhalten eines Objekts bestimmen.

Ein Objekt ist eine Instanz einer Klasse. Es enthält die in der Klasse definierten Attribute und Methoden. Jedes Objekt hat einen eigenen Zustand, der durch seine Attribute repräsentiert wird, und es kann Verhalten durch seine Methoden ausführen.

**Klassen und Objekte in Python:**

In Python wird eine Klasse mit dem Schlüsselwort class definiert, gefolgt vom Namen der Klasse und einem Doppelpunkt. Die Attribute und Methoden der Klasse werden in den folgenden eingerückten Zeilen definiert.

Ein Objekt wird erstellt, indem der Name der Klasse wie eine Funktion aufgerufen wird. Sie können auf die Attribute und Methoden eines Objekts zugreifen, indem Sie den Namen des Objekts und den Namen des Attributs oder der Methode mit einem Punkt (.) dazwischen verwenden.

# Definieren einer Klasse  
class Tier:  
 # Initialisierungsmethode mit Attributen  
 def \_\_init\_\_(self, name, art):  
 self.name = name  
 self.art = art  
  
 # Methode der Klasse  
 def vorstellen(self):  
 return f"Hallo, mein Name ist {self.name} und ich bin ein {self.art}."  
  
# Erstellen eines Objekts der Klasse Tier  
mein\_tier = Tier("Felix", "Hund")  
  
# Zugriff auf Attribute  
print(mein\_tier.name) # Ausgabe: Felix  
print(mein\_tier.art) # Ausgabe: Hund  
  
# Aufruf einer Methode  
print(mein\_tier.vorstellen()) # Ausgabe: Hallo, mein Name ist Felix und ich bin ein Hund.

In diesem Beispiel ist Tier die Klasse, die zwei Attribute (name und art) und eine Methode (vorstellen) hat. mein\_tier ist ein Objekt der Klasse Tier. Wir können auf die Attribute des Objekts zugreifen und seine Methoden aufrufen, indem wir den Namen des Objekts und den Namen des Attributs oder der Methode mit einem Punkt (.) dazwischen verwenden.

**Anwendung auf World of Warcraft (WoW):**

In WoW könnte eine Klasse eine Charakterklasse wie “Jäger” oder “Magier” repräsentieren, und ein Objekt wäre ein spezifischer Charakter, der von einem Spieler erstellt wurde. Die Klasse definiert die Fähigkeiten und Eigenschaften, die alle Charaktere dieser Klasse gemeinsam haben, und das Objekt repräsentiert einen Charakter mit seinem eigenen Zustand und Verhalten.

class Charakter:  
 def \_\_init\_\_(self, klasse, rasse, name, level):  
 self.klasse = klasse  
 self.rasse = rasse  
 self.name = name  
 self.level = level  
  
 def angreifen(self):  
 print(f"Der {self.klasse} greift an!")  
  
mein\_charakter = Charakter("Jäger", "Nachtelf", "Lêgolúlz", 60)  
mein\_charakter.angreifen() # Ausgabe: Der Jäger greift an!

In diesem Beispiel ist Charakter die Klasse, mein\_charakter ist ein Objekt dieser Klasse, und angreifen ist eine Methode, die das Verhalten des Charakters repräsentiert.

## Abstraktion

Abstraktion ist ein grundlegendes Konzept der objektorientierten Programmierung (OOP). Es bezieht sich auf die Fähigkeit, komplexe Realitätsaspekte in einfache Modelle zu überführen. Dabei werden nur die relevanten Daten und Methoden für eine bestimmte Aufgabe berücksichtigt, während unwichtige Details ausgeblendet werden. Dies erleichtert das Verständnis und die Handhabung von komplexen Systemen.

Stellen Sie sich vor, Sie fahren ein Auto. Um das Auto zu fahren, müssen Sie wissen, wie man das Lenkrad dreht, das Gaspedal und die Bremse bedient. Aber Sie müssen nicht wissen, wie der Motor genau funktioniert, wie das Getriebe die Räder antreibt, wie die Elektronik im Auto funktioniert, usw. Diese Details sind abstrahiert, d.h. sie sind verborgen, weil Sie sie nicht wissen müssen, um das Auto zu fahren.

In der Programmierung erreichen wir Abstraktion durch die Verwendung von Klassen und Objekten. Eine Klasse definiert, welche Daten und Methoden ein Objekt haben wird, aber sie verbirgt die genaue Implementierung dieser Methoden.

thrall = Schamane("Thrall", 70)  
jaina = Magier("Jaina", 70)  
  
print(thrall.aktion()) # Ausgabe: Thrall greift mit dem Hammer an!  
print(jaina.aktion()) # Ausgabe: Jaina wirkt einen Zauber!

Obwohl wir nicht wissen, wie die aktion Methode in den Klassen Schamane und Magier implementiert ist, können wir sie immer noch verwenden, weil die Details abstrahiert sind.

## Vererbung

Vererbung ist ein Konzept der objektorientierten Programmierung (OOP), das es ermöglicht, eine neue Klasse auf Basis einer bestehenden Klasse zu erstellen. Die neue Klasse, genannt Unterklasse oder abgeleitete Klasse, erbt die Eigenschaften und Methoden der bestehenden Klasse, die als Oberklasse oder Basisklasse bezeichnet wird. Dies fördert die Wiederverwendbarkeit und Strukturierung des Codes.

**Vererbung in Python**

In Python wird Vererbung durch die Definition einer neuen Klasse erreicht, die eine bestehende Klasse als Basis verwendet. Dies geschieht durch die Angabe der Basisklasse in Klammern nach dem Klassennamen.

class Tier:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def sprich(self):  
 pass  
  
class Hund(Tier):  
 def sprich(self):  
 return f"{self.name} sagt: Wuff!"  
  
class Katze(Tier):  
 def sprich(self):  
 return f"{self.name} sagt: Miau!"

nehmen wir zur weiteren Veranschaulichung nochmal ein Beispiel aus WoW

#Basisklasse:  
  
class Charakter:  
 def \_\_init\_\_(self, name, level):  
 self.name = name  
 self.level = level  
  
 def info(self):  
 return f"{self.name}, Level {self.level}"

#Abgeleitete Klassen  
  
class Schamane(Charakter):  
 def kampfschrei(self):  
 return f"{self.name} ruft: Für die Horde!"  
   
  
class Magier(Charakter):  
 def zauber(self):  
 return f"{self.name} zaubert: Arkanschlag!"

#Anwendung  
  
thrall = Schamane("Thrall", 70)  
jaina = Magier("Jaina", 70)  
  
print(thrall.info()) # Thrall, Level 70  
print(thrall.kampfschrei()) # Thrall ruft: Für die Horde!  
print(jaina.info()) # Jaina, Level 70  
print(jaina.zauber()) # Jaina zaubert: Arkanschlag!

In diesem Beispiel erben die Klassen Schamane und Magier von der Basisklasse Charakter und fügen spezifische Methoden hinzu.

## Polymorphismus

Polymorphismus ist ein weiteres grundlegendes Konzept der objektorientierten Programmierung (OOP). Es ermöglicht, dass Methoden unterschiedlich arbeiten, je nachdem, welcher Objekttyp sie aufruft. Mit anderen Worten, eine einzige Methode oder Funktion kann auf verschiedene Weisen verwendet werden, abhängig von der Anzahl und Art der Argumente oder dem Typ des Objekts, auf das sie angewendet wird.

**Polymorphismus in Python:**

In Python wird Polymorphismus durch die Verwendung von Vererbung und Methodenüberschreibung erreicht. Wenn eine Unterklasse eine Methode der Oberklasse überschreibt, kann die Methode unterschiedliche Aktionen ausführen, je nachdem, ob sie auf ein Objekt der Oberklasse oder der Unterklasse angewendet wird, obwohl der Name der Methode derselbe ist.

class Tier:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def sprich(self):  
 pass  
  
class Hund(Tier):  
 def sprich(self):  
 return f"{self.name} sagt: Wuff!"  
  
class Katze(Tier):  
 def sprich(self):  
 return f"{self.name} sagt: Miau!"  
  
def tier\_stimme(tier):  
 print(tier.sprich())  
  
hund = Hund("Rex")  
katze = Katze("Felix")  
  
tier\_stimme(hund) # Ausgabe: Rex sagt: Wuff!  
tier\_stimme(katze) # Ausgabe: Felix sagt: Miau!

In diesem Beispiel haben wir eine Funktion tier\_stimme, die ein Objekt der Klasse Tier als Argument nimmt und die Methode sprich dieses Objekts aufruft. Obwohl die Funktion tier\_stimme immer die gleiche Methode sprich aufruft, führt sie unterschiedliche Aktionen aus, je nachdem, ob das übergebene Objekt ein Hund oder eine Katze ist. Das ist ein Beispiel für Polymorphismus.

**Anwendung auf WoW**

In unserem World of Warcraft Beispiel könnten wir eine Methode namens aktion in der Basisklasse Charakter definieren und diese Methode in den abgeleiteten Klassen Schamane und Magier überschreiben, um unterschiedliche Aktionen auszuführen:

class Charakter:  
 def \_\_init\_\_(self, name, level):  
 self.name = name  
 self.level = level  
  
 def info(self):  
 return f"{self.name}, Level {self.level}"  
  
 def aktion(self):  
 pass  
  
class Schamane(Charakter):  
 def aktion(self):  
 return f"{self.name} greift mit dem Hammer an!"  
  
class Magier(Charakter):  
 def aktion(self):  
 return f"{self.name} wirkt einen Zauber!"

Jetzt, wenn wir die aktion Methode auf verschiedene Charakterobjekte anwenden, erhalten wir unterschiedliche Ergebnisse, je nachdem, ob das Objekt ein Schamane oder ein Magier ist:

thrall = Schamane("Thrall", 70)  
jaina = Magier("Jaina", 70)  
  
print(thrall.aktion()) # Ausgabe: Thrall greift mit dem Hammer an!  
print(jaina.aktion()) # Ausgabe: Jaina wirkt einen Zauber!

Das ist Polymorphismus in Aktion! Obwohl die Methode aktion sowohl in der Schamane als auch in der Magier Klasse definiert ist, führt sie unterschiedliche Aktionen aus, je nachdem, auf welchen Charaktertyp sie angewendet wird.

## Kapselung

* **Definition**: Kapselung ist das Verbergen der internen Zustände und das Einschränken des Zugriffs auf die Attribute und Methoden einer Klasse, um die Datenintegrität zu gewährleisten.
* **Zweck**: Sie schützt die Daten vor unbefugtem Zugriff und Manipulation und ermöglicht eine kontrollierte Schnittstelle zur Interaktion mit dem Objekt.

**Kapselung in Python:**

* **Private Attribute**: In Python werden private Attribute durch ein führendes Unterstrich (\_) gekennzeichnet. Diese Attribute sollten nicht direkt von außerhalb der Klasse zugegriffen werden.
* **Getter und Setter**: Methoden, die den Zugriff auf private Attribute ermöglichen und kontrollieren.

**Beispiel in Python:**

class Konto:  
 def \_\_init\_\_(self, kontonummer, saldo):  
 self.\_kontonummer = kontonummer # private Attribut  
 self.\_saldo = saldo # private Attribut  
  
 def get\_saldo(self):  
 return self.\_saldo  
  
 def set\_saldo(self, betrag):  
 if betrag >= 0:  
 self.\_saldo = betrag  
 else:  
 print("Ungültiger Betrag")  
  
# Verwendung  
konto = Konto("123456", 1000)  
print(konto.get\_saldo()) # Ausgabe: 1000  
konto.set\_saldo(1500)  
print(konto.get\_saldo()) # Ausgabe: 1500  
konto.set\_saldo(-500) # Ausgabe: Ungültiger Betrag

**Kapselung in WoW**

In WoW haben Charaktere Gesundheitspunkte (HP), die ihre aktuelle Vitalität repräsentieren. Diese Gesundheitspunkte sind ein gutes Beispiel für ein Attribut, das gekapselt werden sollte, um sicherzustellen, dass es nicht auf einen ungültigen Wert gesetzt wird.

class Charakter:  
 def \_\_init\_\_(self, name, max\_hp):  
 self.\_name = name  
 self.\_max\_hp = max\_hp  
 self.\_current\_hp = max\_hp  
  
 def get\_hp(self):  
 return self.\_current\_hp  
  
 def take\_damage(self, damage):  
 if damage < 0:  
 print("Ungültiger Schaden")  
 else:  
 self.\_current\_hp -= damage  
 if self.\_current\_hp < 0:  
 self.\_current\_hp = 0  
 print(f"{self.\_name} hat {damage} Schaden genommen!")  
  
 def heal(self, hp):  
 if hp < 0:  
 print("Ungültige Heilung")  
 else:  
 self.\_current\_hp += hp  
 if self.\_current\_hp > self.\_max\_hp:  
 self.\_current\_hp = self.\_max\_hp  
 print(f"{self.\_name} wurde um {hp} HP geheilt!")  
  
# Verwendung  
thrall = Charakter("Thrall", 100)  
print(thrall.get\_hp()) # Ausgabe: 100  
thrall.take\_damage(45)  
print(thrall.get\_hp()) # Ausgabe: 55  
thrall.heal(30)  
print(thrall.get\_hp()) # Ausgabe: 85  
thrall.heal(30)  
print(thrall.get\_hp()) # Ausgabe: 100

In diesem Beispiel ist Charakter die Klasse, die einen WoW-Charakter repräsentiert. Die Methoden take\_damage und heal ändern die aktuellen HP des Charakters, stellen aber sicher, dass die HP nicht auf einen ungültigen Wert gesetzt werden. Das ist ein Beispiel für Kapselung.