Einführung in Scala

Wie werden veränderliche und unveränderliche Variablen in Scala definiert?

- immutable mit **val**:

```
val x = 5
```

- mutable mit **var**:

```
var y = 10
```

Wie werden Methoden deklariert?

- Methoden werden mit dem Schlüsselwort **def** deklariert:

```
def max(list : List[Int]) : Int = {
    var m = Integer.MIN_VALUE
    for (x <- list) {
        if (x > m) then m = x
    }
    m
}
```

Wie werden Klassen deklariert?

- Klassen werden mit dem Schlüsselwort **class** deklariert:

```
class Person:
    ...
```

oder mit Klammern:

```
class Person {
    ...
}
```

- Mit Erbschaft:

```
class Student extends Person:
    ...
    override def toString : String = ...
```

Was sind traits und wie werden sie verwendet?

- traits sind wie Interfaces mit default Implementierungen:

```
trait Writeable :
    def write(out: PrintStream) : Unit
    def writeln(out: PrintStream) : Unit = {
        write(out)
        out.println()
    }
```

Was sind objects und wie werden sie verwendet?

- objects sind Singleton-Instanzen einer Klasse:

```
object HelloWorld extends App :
    println("Hallo World")
```

Was ist Type Inference?

- Scala kann Typen automatisch inferieren:

```
val list = List(2, 1, 4, 2) // List[Int]
```

Wie werden Generics in Scala verwendet?

- Generics werden mit eckigen Klammern definiert:

```
class Box[T](val value: T)
```

- Methoden:

```
def compose[A, B, C](f : A => B, g : B => C) : A => C =
    x => g(f(x))
```

-> nimmt zwei Funktionen als Parameter und gibt eine neue Funktion zurück. f wandelt einen Wert von Typ A in einen Wert von Typ B um, g wandelt einen Wert von Typ B in einen Wert von Typ C um. Die zusammengesetzte Funktion wandelt einen Wert von Typ A in einen Wert von Typ C um, indem sie f auf x anwendet und dann g auf das Ergebnis anwendet.

Was ist ein Companion Object?

- Ein Singelton-Objekt mit dem selben Namen wie eine Klasse
- muss in der selben Datei wie die Klasse definiert werden
- kann auf private Elemente der Klasse zugreifen
- wird verwendet, um statische Methoden und Felder (gehören zur Klasse, nicht zur Instanz) zu definieren

```
class Car(model: String, initial: Int, mileage: Double)
    extends Vehicle(model, initial) { ...}

object Car {
    var carPool: List[Car] = List()

    def apply(model: String, year: Int, mileage: Double) = {
        val car = new Car(model, year, mileage)
        carPool = car :: carPool
        car
    }
}
```

```
Car("Opel", 45000, 53.1)
Car("BMW", 0, 62.4)
val car1 = Car.carPool(0)
val car2 = Car.carPool(1)
```

Wie wird multiple Vererbung in Scala realisiert?

- Traits können mehrfach vererbt werden:

```
trait Person extends AnyRef
trait Friend extends AnyRef
trait Professional extends Person
trait Student extends Professional
trait HasGender extends Person
trait Male extends HasGender

class StudentFriend extends Student with Male with Friend
```

Wie werden Einschränkungen für generische Typen in Scala definiert?

- mit Type Bounds
  - Upper Bound mit <:
  - Lower Bound mit >:

```
class Animal {
  def speak(): Unit = println("Some sound")
}

class Dog extends Animal {
  override def speak(): Unit = println("Woof!")
}

class Cat extends Animal {
  override def speak(): Unit = println("Meow!")
}

// Funktion, die nur Untertypen von Animal akzeptiert
def makeAnimalSpeak[T <: Animal](animal: T): Unit = animal.speak()

makeAnimalSpeak(new Dog)  // Woof!
makeAnimalSpeak(new Cat)  // Meow!
// makeAnimalSpeak("Hello")  ❌ Fehler! String ist kein Animal
```

```
class Animal
class Dog extends Animal
class Puppy extends Dog

// Funktion, die nur Obertypen von Dog akzeptiert
def adopt[T >: Dog](animal: T): Unit = println("Animal adopted!")

adopt(new Animal)  // ☑ Animal ist ein Obertyp von Dog
adopt(new Dog)     // ☑ Dog ist genau Dog
// adopt(new Puppy)  ❌ Fehler! Puppy ist ein Untertyp, kein Obertyp von Dog
```

Was ist Use-Site Variance und wie wird es in Scala verwendet?

Use-Site Variance in Scala bezieht sich auf die Verwendung von Variance (+T, -T) nicht in der Klassendefinition (Definition-Site Variance), sondern dort, wo ein Typ tatsächlich verwendet wird.

- Kovarianz: `+T` -> C[T] ist ein Untertyp von C[U], wenn T ein Untertyp von U ist
- Kontravarianz: `-T` -> C[T] ist ein Untertyp von C[U], wenn T ein Obertyp von U ist
- Invarianz: keine Vererbung zwischen C[T] und C[U]

Definition-Site Variance:

```scala
class Box[+T]  // Kovariant: Box[Dog] <: Box[Animal]
class Printer[-T] // Kontravariant: Printer[Animal] <: Printer[Dog]
```

Use-Site Variance:

```scala
def printAnimals[T <: Animal](animals: List[T]): Unit =
  animals.foreach(_.speak())
```

-> T wird zur Laufzeit als irgendein Untertyp von Animal behandelt

```scala
val students: ListBuffer[Student] = ListBuffer()
val persons: ListBuffer[_ <: Person] = students

persons += new Student()   // ☑ Erlaubt, weil Student <: Person
persons += new Person()    // ❌ Fehler! Könnte falscher Typ sein
persons += new Teacher()   // ❌ Fehler! Teacher könnte inkompatibel sein
```

-> _ <: Person erlaubt das Lesen als Person, aber schränkt das Schreiben ein