

# SOLID Designprinciples


---

- SOLID: Ein Akronym für fünf grundlegende Prinzipien der objektorientierten Programmierung und des Designs.
- Ziel: Förderung von wartbarem, verständlichem und flexiblem Code.



## Single Responsibility Principle (SRP)

---

- Definition: Eine Klasse sollte nur einen einzigen Grund zur Änderung haben.
  - Wichtigkeit: Vereinfacht Wartung und Testbarkeit.
  - Beispiel: Trennung von Datenzugriffs - und Geschäftslogik (data -, service layer/packages)
- 

# Liskov's Substitution Principle (LSP)


---

- Definition: Objekte einer Superklasse sollten durch Objekte von Subklassen ersetzbar sein, ohne die Korrektheit des Programms zu beeinträchtigen.
- Bedeutung: Sicherstellung der Konsistenz und Austauschbarkeit von Klassen.
- Beispiel: Vermeidung der Änderung erwarteter Verhaltensweisen in Subklassen durch „falscher“ Überschreibung von Funktionen.



# Liskov's Substitution Principle (LSP)

---

- Definition: Objekte einer Superklasse sollten durch Objekte von Subklassen ersetzbar sein, ohne die Korrektheit des Programms zu beeinträchtigen.
  - Bedeutung: Sicherstellung der Konsistenz und Austauschbarkeit von Klassen.
  - Beispiel: Vermeidung der Änderung erwarteter Verhaltensweisen in Subklassen durch „falscher“ Überschreibung von Funktionen.
- 

# Dependency Inversion Principle (DIP)

---

- „Higher level modules should not depend on lower level modules, but rather on their abstraction“
- Definition: Abhängigkeiten sollten auf Abstraktionen beruhen, nicht auf Konkretisierungen.
- Ziel: Reduzierung der Kopplung und Erhöhung der Modularität.
- Beispiel: Verwendung von Interface -Injektion oder Konstruktoren -Injektion für Abhängigkeiten.

## Generics

---



- Generics wurden in Java eingeführt, um ClassCastExceptions zu vermeiden
- Mit Generics ist es möglich einen Datentyp bereits zur Kompilierungszeit festzulegen.
- Generics machen es möglich Klassen, Variablen und Funktionen für beliebige Datentypen zu entwickeln (also eine Implementierung für mehrere Datentypen)
- Generics wurden mit Java 5 eingeführt.

# Generische Wildcards



- Nehmen wir an man wolle nun Listen von Integer oder Typen übergeben, die Überklassen vom Typ Integer sind, zum Beispiel Number oder Object.
- Dann müsste man ein lower Bound angeben (mindestens Integer und alles was darüber liegt)

```
public static double sumWorkingWithDiffDatatypes(List<? super Integer> list){...
```

# Generische Wildcards

- Für obigen Fall verwendet man Generische Wildcards (?), um verschiedene Datentypen zu ermöglichen.

```
public static double sumWorkingWithDiffDatatypes(List<? extends Number> list) {  
    double sum = 0;  
    for(Number n : list){  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

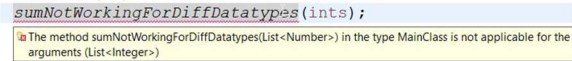
- Man könnte in den spitzen Klammern auch nur <?> angeben (unbound), dann gäbe es keine Upper Bound Class und man könnte noch mehr Datentypen übergeben.

# Generische Wildcards

---

- Obige Funktion würde einen Compilerfehler erzeugen, wenn man zum Beispiel eine Liste von Integer übergibt.

```
List<Integer> ints = new ArrayList<>();
ints.add(10);
ints.add(10);
ints.add(10);
sumNotWorkingForDiffDatatypes(ints);
```



# Generische Wildcards

---

Nehmen wir an wir möchten eine Funktion schreiben, die für eine Liste von Zahlen beliebigen Datentyps alle Werte addiert. Dann würde folgendes nicht funktionieren:

```
// This method is not working for List of Integer or Double
public static double sumNotWorkingForDiffDatatypes(List<Number> list) {
    double sum = 0;
    for(Number n : list){
        sum += n.doubleValue();
    }
    return sum;
}
```

# Subklassen von Generischen Klassen

---

- Subklassen können von generischen Klassen abgeleitet werden und mehrere generische Typen enthalten, wichtig ist dabei, dass der ursprüngliche Subtyp erhalten bleibt:

```
public interface MyList<E,T> extends List<E>{...}
```

## Generische Methoden für bestimmte Typen

---

- Es gibt auch die Möglichkeit, dass eine generische Funktion für bestimmte Typen nur Typen zulässt, die von mehreren Bindungen abhängen .
- In diesem Fall sind die Typen mit einem „&“ verknüpft, beim ersten Typ darf es sich um eine Klasse handeln, die anderen müssen Interfaces sein.

```
public static <T extends Dog & Animal> void letThemTalk(T anim)
{
    anim.TalkToMe();
}
```

# Generische Methoden für bestimmte Typen

---

```
public static <T extends Animal> boolean compare(T t1, T t2){  
    return t1.getClass().toString().equals(t2.getClass().toString());  
}
```

```
Animal whiskey = new Dog();  
Dog rintintin = new Dog();  
boolean sameClass = compare(whiskey, rintintin);  
// Welchen Output wird folgende Zeile liefern?  
System.out.println(sameClass ? "Die beiden sind gleich!" : "Die  
beiden sind ungleich!");
```

## Generische Methode für bestimmte Typen

---

- Manchmal möchte man bestimmte Funktionen nur für bestimmte Datentypen erlauben.
- In diesem Fall definiert man einen sogenannten **Bounded Parameter**.
- Einen Bounded Parameter definiert man mit dem Typnamen, gefolgt von einem „extends“ und der Klasse, die diese Funktion akzeptiert (den Upper Bound = die „höchste“ Klasse)

# Generische Methoden

---

```
public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T>
g2)
{
    return g1.get().equals(g2.get());
}
```

```
boolean areGenericsEqual = isEqual(genType, genType2);
```

