

# Funktionale und objektorientierte Programmierkonzepte



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Präsenz-Sprechstunde B

Simon Hock, Nhan Huynh, Daniel Mangold



Generics [5min]

Kovarianz [10min]

    Rückgabewert

    Arrays

Wildcards [10min]

    Lower bounded Wildcard

    Upper bounded Wildcard

Arbeitsphase [75min]

- Generizität nur auf **Referenztypen** möglich!
- Abstraktion, Wiederverwendbarkeit von Klassen bzw. Methoden
- Weniger redundanter Code, da man eine Klasse/ Methode durch Generizität auf mehrere Referenztypen anwenden kann, ohne diese speziell für einen Typ zu spezifizieren.



- Beruhen auf dem Ersetzbarkeitsprinzip
- Der zurückgelieferte Wert der Unterklassen muss mit dem der Oberklasse vereinbar sein, also es darf kein allgemeinerer Typ sein
- Wo kann das auftreten?
  - ▣ Parameter
  - ▣ Rückgabetyt
  - ▣ `throws`-klausel

- Der aktuelle Rückgabetyt entspricht entweder dem formalen Rückgabetyt oder ist ein Subtyp vom formalen Rückgabetyt
  - ▢ Rückgatetyt  $\neq$  Rückgabewert

## Rückblick:

- *Statischer Typ:*
  - ▢ Typ, der bei der Variablendeklaration angegeben wird
  - ▢ Ist bei der Kompilierung bekannt
- *Dynamischer Typ:*
  - ▢ Typ des tatsächlichen Objekts
  - ▢ Ist erst zur Laufzeit bekannt
  - ▢ Kann vom statischen Typ abweichen  $\rightarrow$  Gleich oder Subtyp des statischen Typs

```
1 public interface NumberAdder {  
2  
3     Number add(Number a, Number b);  
4 }
```

```
1 public class IntegerAdder implements NumberAdder {
2
3     @Override
4     public Integer add(Number a, Number b) {
5         return a.intValue() + b.intValue();
6     }
7 }
8
9 public class DoubleAdder implements NumberAdder {
10     @Override
11     public Number add(Number a, Number b) {
12         return a.doubleValue() + b.doubleValue();
13     }
14 }
```

1. Welchen Typ hat die Zeile 7?
2. Welchen Typ hat die Zeile 8?

```
1 NumberAdder adder1 = new IntegerAdder();
2 IntegerAdder adder2 = new IntegerAdder();
3
4 Number a = 5.312; // Type Double
5 Number b = 2; // Type Integer
6
7 adder1.add(a,b);
8 adder2.add(a,b);
```

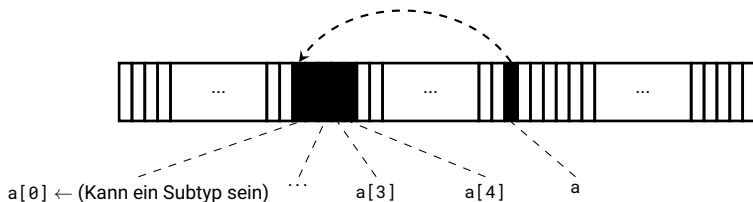


1. Welchen Typ hat die Zeile 7?
2. Welchen Typ hat die Zeile 8?

```
1 NumberAdder adder1 = new IntegerAdder();
2 IntegerAdder adder2 = new IntegerAdder();
3
4 Number a = 5.312; // Type Double
5 Number b = 2; // Type Integer
6
7 adder1.add(a,b);
8 adder2.add(a,b);
```

1. Number
2. Integer

- Die Komponententypen im Array können gleich dem Komponententypen oder ein Subtyp dessen sein
- Der dynamische Typ des Arrays kann gleich dem statischen Typ oder ein Subtyp dessen sein



**Abbildung:** Abstrakte Visualisierung des Speicherplatzes eines Arrays  $a$

# Beispiel: Komponententypen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 byte b = 1;  
2 short s = 3;  
3 int i = 4;  
4 float f = 5.6f;  
5 long l = 7;  
6 double d = 8.9;  
7 Number[] numbers = new Number[6];  
8 numbers[0] = b; numbers[1] = s;  
9 numbers[2] = i; numbers[3] = f;  
10 numbers[4] = l; numbers[5] = d;
```

# Beispiel: Dynamischer Typ



```
1 Number[] integers = new Integer[5];
2 Number[] doubles = new Double[integers.length];
3 for (int i = 0; i < integers.length; i++) {
4     integers[i] = i;
5     doubles[i] = (double) i;
6 }
```

Wirft die Zeile 5 einen Fehler? (Begründung)

Wenn ja:

1. Welcher Fehler wird geworfen?
2. Wird der Fehler während der Kompilierung oder erst zur Laufzeit erkannt?

```
1 Number[] integers = new Integer[5];
2 Number[] doubles = new Double[integers.length];
3 for (int i = 0; i < integers.length; i++) {
4     integers[i] = i;
5     doubles[i] = i;
6 }
```

Wirft die Zeile 5 einen Fehler? (Begründung)

Wenn ja:

1. Welcher Fehler wird geworfen?
2. Wird der Fehler während der Kompilierung oder erst zur Laufzeit erkannt?

```
1 Number[] integers = new Integer[5];
2 Number[] doubles = new Double[integers.length];
3 for (int i = 0; i < integers.length; i++) {
4     integers[i] = i;
5     doubles[i] = i;
6 }
```

1. ArrayStoreException
2. Der Fehler wird erst zur Laufzeit erkannt, da der Komponententyp ein Subtyp des statischen Types des Arrays ist



- Wie sieht es mit Wildcards aus?
- Die Typparameter sind fest d.h. es ist nicht erlaubt, anstelle des eigentlichen Typparameters jetzt andere direkt oder indirekt abgeleitete Klassen einzusetzen
  - ▣ Nichtübertragbarkeit von Vererbung auf Typparameter
- „Die Typparameter werden vom Compiler fest in die Klasse gebrannt“

```
1 public class Printer<T> {  
2  
3     public void print(T element) {  
4         System.out.println("Printed element: " + element.toString());  
5     }  
6 }
```

```
1 Printer<Number> printer = new Printer<Number>();  
2 printer = new Printer<Integer>(); // Compile-Error
```



- Ermöglicht Kovarianz bei Typparametern
- Definition der Einschränkungen bei der **Instanziierung** von Typparametern!
- Flexibilität von Typparameter von Objekten
- Sei X eine generische Klasse mit einem Typparameter T:
  - ▣ Nach unten beschränkt:  $X<? \text{super } T>$  – *Lower bounded Wildcard*
  - ▣ Nach oben beschränkt:  $X<? \text{extends } T>$  – *Upper bounded Wildcard*

```
1 Printer<? extends Number> printer = new Printer<Number>();  
2 printer = new Printer<Integer>();  
3 printer = new Printer<Double>();
```



- Typparameter nach unten in der Vererbungshierarchie beschränkt
- Typparameter ist entweder
  - ▣ T oder
  - ▣ direkt oder indirekte Oberklassen von T oder
  - ▣ direkt oder indirekte implementierte Interfaces von T
- Formaler Aufbau:  $X < ? \text{ super } T >$  (X = generische Klasse)

```
1 public class Animal {}
2 public class Cat extends Animal{}
3 public class RagdollCat extends Cat {}
4 public class MaineCoonCat extends Cat {}
5 public class BlackMaineCoonCat extends MaineCoonCat {}
```

Wir möchten eine schwarze Maine Coon Katze in einer Liste hinzufügen:

```
1 public static void addCat(List<BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat);  
4 }
```

Wir möchten eine schwarze Maine Coon Katze in einer Liste hinzufügen:

```
1 public static void addCat(List<BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat);  
4 }
```

Das erlaubt uns nur das Hinzufügen einer schwarzen Maine Coon Katze in einer `List<BlackMaineCoonCat>`. Kann man das nicht flexibler machen?

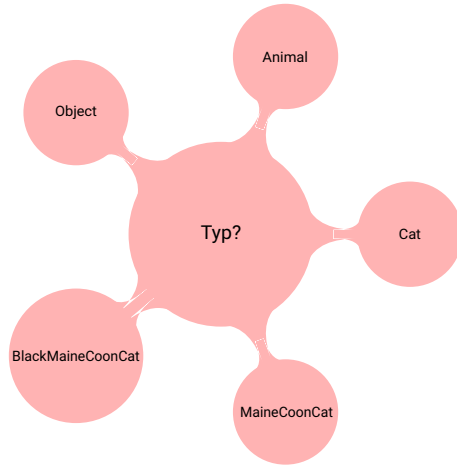
```
1 public static void addCat(List<? super BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat);  
4 }
```

```
1 List<BlackMaineCoonCat> b1 = new ArrayList<>();
2 List<MaineCoonCat> m1 = new ArrayList<>();
3 List<Cat> c1 = new ArrayList<>();
4 List<Animal> a1 = new ArrayList<>();
5 List<Object> o1 = new ArrayList<>();
6 BlackMaineCoonCat cat = new BlackMaineCoonCat();
7
8 addCat(b1, cat);
9 addCat(m1, cat);
10 addCat(c1, cat);
11 addCat(a1, cat);
12 addCat(o1, cat);
```



Welchen Typ können wir in der Zeile 4 lesen?

```
1 public static void addCat(List<? super BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat);  
4     cats.get(0); // Type?  
5 }
```



# Welchen Typ können wir in allen Fällen garantieren?

Alle Klassen erben direkt oder indirekt von der Klasse `Object`, also können wir mindestens den Typ `Object` garantieren!

```
1 public static void addCat(List<? super BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat);  
4     Object o = cats.get(0);  
5 }
```

Welche Zeilen laufen fehlerfrei durch?

```
1 public static void addCat(List<? super BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat); // Pass or error?  
4     cats.add(new MaineCoonCat()); // Pass or error?  
5     cats.add(new Cat()); // Pass or error?  
6     cats.add(new Animal()); // Pass or error?  
7     cats.add(new Object()); // Pass or error?  
8 }
```

---

Listentyp ist entweder:

- Tatsächlicher Listentyp: `BlackMaineCoonCat`
  - ▢ Kann nur `BlackMaineCoonCat` oder Subtypen dessen einfügen
  - ▢ Fehlerhafte Zeilen: 4, 5, 6,7
- Tatsächlicher Listentyp: `MaineCoonCat`
  - ▢ Kann nur `MaineCoonCat` oder Subtypen dessen einfügen
  - ▢ Fehlerhafte Zeilen: 5, 6,7
- Tatsächlicher Listentyp: `Cat`
  - ▢ Kann nur `Cat` oder Subtypen dessen einfügen
  - ▢ Fehlerhafte Zeilen: 6, 7
- Tatsächlicher Listentyp: `Animal`
  - ▢ Kann nur `Animal` oder Subtypen dessen einfügen
  - ▢ Fehlerhafte Zeilen: 7
- Tatsächlicher Listentyp: `Object`
  - ▢ Kann nur `Object` oder Subtypen dessen einfügen
  - ▢ Fehlerhafte Zeilen: Keine

# Welchen Typ können wir in allen Fällen garantieren?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wir können in jeder Liste eine `BlackMaineCoonCat` oder ein Subtypen dessen auf jeden Fall hinzufügen!

```
1 public static void addCat(List<? super BlackMaineCoonCat> cats,  
2     BlackMaineCoonCat cat) {  
3     cats.add(cat); // Pass  
4     cats.add(new MaineCoonCat()); // Error  
5     cats.add(new Cat()); // Error  
6     cats.add(new Animal()); // Error  
7     cats.add(new Object()); // Error  
8 }
```



- Typparameter nach oben in der Vererbungshierarchie beschränkt.
- Typparameter ist entweder
  - ▣ T oder
  - ▣ direkt oder indirekte Unterklasse von T
- Formaler Aufbau: `X<? extends T>` (X = generische Klasse)
- `X<?> = X<? extends Object>`

Wir möchten die Summe von Zahlen in einer Liste bilden:

```
1 public static double sum(List<Number> numbers) {  
2     double sum = 0;  
3     for (Number number : numbers) {  
4         sum += number.doubleValue();  
5     }  
6     return sum;  
7 }
```



Wir möchten die Summe von Zahlen in einer Liste bilden:

```
1 public static double sum(List<Number> numbers) {  
2     double sum = 0;  
3     for (Number number : numbers) {  
4         sum += number.doubleValue();  
5     }  
6     return sum;  
7 }
```

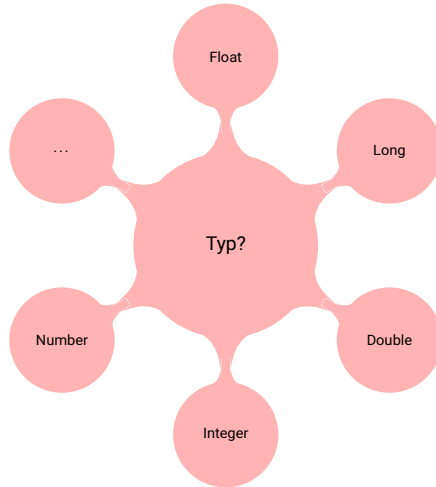
Das erlaubt uns nur das Bilden einer Summe für den Typparameter `Number`. Was machen wir, wenn wir die Summe von `Integer` oder `Double` bilden wollen, ohne redundanten Code zu schreiben?

```
1 public static double sum(List<? extends Number> numbers) {  
2     double sum = 0;  
3     for (Number number : numbers) {  
4         sum += number.doubleValue();  
5     }  
6     return sum;  
7 }
```

```
1 List<Number> n1 = new ArrayList<>();
2 List<Integer> i1 = new ArrayList<>();
3 List<Double> d1 = new ArrayList<>();
4 List<Long> l1 = new ArrayList<>();
5 List<Float> f1 = new ArrayList<>();
6 sum(n1);
7 sum(i1);
8 sum(d1);
9 sum(l1);
10 sum(f1);
```

Welchen Typ können wir in der Zeile 3 lesen? (Außer Number)

```
1 public static double sum(List<? extends Number> numbers) {  
2     double sum = 0;  
3     for (Number number : numbers) {  
4         sum += number.doubleValue();  
5     }  
6     return sum;  
7 }
```



# Welchen Typ können wir in allen Fällen garantieren?



Alle Klassen sind Subtypen von Number, also können wir mindestens den Typ Number garantieren!

```
1 public static double sum(List<? extends Number> numbers) {  
2     double sum = 0;  
3     for (Number number : numbers) {  
4         sum += number.doubleValue();  
5     }  
6     return sum;  
7 }
```

Welche Zeilen laufen fehlerfrei durch?

```
1 public static double sum(List<? extends Number> numbers) {  
2     ...  
3     Number number = 1; Integer integer = 2;  
4     Double d = 3.4;  
5     numbers.add(number); // Pass or Error?  
6     numbers.add(integer); // Pass or Error?  
7     numbers.add(d); // Pass or Error?  
8     return sum;  
9 }
```

---

Listentyp ist entweder:

- Tatsächlicher Listentyp: Number
  - ▣ Kann Number oder ein Subtyp dessen einfügen
  - ▣ Fehlerhafte Zeilen: Keine
- Tatsächlicher Listentyp: Integer
  - ▣ Kann Integer oder ein Subtyp dessen einfügen
  - ▣ Fehlerhafte Zeilen: 5, 7
- Tatsächlicher Listentyp: Double
  - ▣ Kann Double oder ein Subtyp dessen einfügen
  - ▣ Fehlerhafte Zeilen: 5, 6
- Tatsächlicher Listentyp: Long
  - ▣ Kann Long oder ein Subtyp dessen einfügen
  - ▣ Fehlerhafte Zeilen: 5, 6, 7
- ...



# Welchen Typ können wir in allen Fällen garantieren?



Wir können in jeder Liste auf jeden Fall `null` hinzufügen!

```
1 public static double sum(List<? extends Number> numbers) {  
2     ...  
3     Number number = 1; Integer integer = 2; Double d = 3.4;  
4     numbers.add(number); // Error  
5     numbers.add(integer); // Error  
6     numbers.add(d); // Error  
7     numbers.add(null); // Pass  
8     return sum;  
9 }
```

Gegeben sei eine Liste `List<E>` mit einem beliebigen Typ `E`.

- Schreiben erfolgt mittels der Methode `add(E e)`
- Lesen erfolgt mittels der Methode `get(int index)`

Statischer Typ der Liste	Dynamischer Typ der Liste	Schreiben	Lesen
<code>List&lt;?&gt;</code>	Object und alle Subklassen von Object	Nur <code>null</code>	Object
<code>List&lt;? super E&gt;</code>	E und alle Basisklassen und implementierten Interfaces von E	E und alle Subtypen von E	Object
<code>List&lt;? extends E&gt;</code>	E und alle Subklassen von E	Nur <code>null</code>	E

Tabelle: Überblick Wildcards

## Selbstständiges Arbeiten