

# FOP Recap #9



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Generics



# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

## Mehrdimensionale Arrays

- Initialisierung

- Zugriff auf Elemente

- Beispiel

Generics-Grundlagen

Diamond operator

Generische Methoden

Eingeschränkte Typparameter

Erben von generischen Klassen

Erstellen von generischen Arrays

Wildcards

Type Erasure



- Haben als Komponententypen wieder Arrays
- Typischerweise 2- oder 3-dimensionale Arrays
- Zugriff über mehrere Indizes
- Wichtig: Subarrays müssen nicht die gleiche Länge haben
- Zusätzliche Kniffe fürs Erstellen

# Mehrdimensionale Arrays

## Initialisierung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 int[][] bigArray = new int[2][5];
```

```
1 int[][] bigArray = new int[2][];  
2 bigArray[0] = new int[5];  
3 bigArray[1] = new int[5];
```

```
1 int[][] bigArray = new int[2][];  
2 bigArray[0] = new int[] {0, 0, 0, 0, 0};  
3 bigArray[1] = new int[] {0, 0, 0, 0, 0};
```

```
1 int[][] bigArray = new int[][] {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}};
```

# Mehrdimensionale Arrays

## Zugriff auf Elemente



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 int[][] bigArray = new int[2][5];
```

```
1 int[] smallArray = bigArray[0];  
2 smallArray[2] = 3;
```

```
1 bigArray[0][2] = 3;
```

# Mehrdimensionale Arrays

## Zugriff auf Elemente



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 int[][] bigArray = new int[2][5];
```

```
1 int[] smallArray = bigArray[0];
```

```
2 int value = smallArray[2];
```

```
1 int value = bigArray[0][2];
```

# Mehrdimensionale Arrays

## Beispiel



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 int[][] array = new int[3][];  
2 array[0] = new int[] {1, 2, 3, 4, 5};  
3 array[1] = new int[] {3, 9};  
4 array[2] = new int[] {5, -11, 8};
```

	0	1	2	3	4
array[0]	1	2	3	4	5
array[1]	3	9			
array[2]	5	-11	8		

# Mehrdimensionale Arrays

## Beispiel



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 int[][] array = ....; // von vorheriger slide übernommen
2 array[0][0] = 10;
3 array[2][1] = -10;
```

	0	1	2	3	4
array[0]	10	2	3	4	5
array[1]	3	9			
array[2]	5	-10	8		



# Mehrdimensionale Arrays

## Beispiel



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 int[][] array = new int[3][];  
2 array[1] = new int[] {3, 9, 10, 25};
```

array[0]	null			
array[1]	3	9	10	25
array[2]	null			

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

- Instanziierung

- Nutzen von Typparameter

- Konventionen für Typparameter-Benennung

- Mehrere Typparameter

Diamond operator

Generische Methoden

Eingeschränkte Typparameter

Erben von generischen Klassen

Erstellen von generischen Arrays

Wildcards

Type Erasure



ohne Generics



```
1 public class StringHolder {  
2     public String value;  
3 }  
4 public class IntegerHolder {  
5     public Integer value;  
6 }
```



mit Generics



```
1 public class Holder<T> {  
2     public T value;  
3 }
```



</>

ohne Generics

</>

```
1 public class StringHolder {  
2     public String value;  
3 }  
4 public class IntegerHolder {  
5     public Integer value;  
6 }
```

</>

mit Generics

</>

```
1 public class Holder<T> {  
2     public T value;  
3 }
```

## Vorteile von Generics

- Lösung mit Generics kürzer

</>

ohne Generics

</>

```
1 public class StringHolder {  
2     public String value;  
3 }  
4 public class IntegerHolder {  
5     public Integer value;  
6 }
```

</>

mit Generics

</>

```
1 public class Holder<T> {  
2     public T value;  
3 }
```

## Vorteile von Generics

- Lösung mit Generics kürzer
- Vermeidet Redundanz  $\Rightarrow$  Fehler vermeiden, Wartbarkeit verbessern



```
1 public class Holder<T> {  
2     public T value;  
3 }
```



```
1 public class Holder<T> {  
2     public T value;  
3 }
```

```
1 Holder<Integer> integerHolder = new Holder<Integer>();  
2 Holder<Boolean> booleanHolder = new Holder<Boolean>();
```



```
1 public class Holder<T> {  
2     public T value;  
3  
4     public T getValue() {  
5         return value;  
6     }  
7     public void killItWithFire() {  
8         value = null;  
9     }  
10 }
```





- Ein Großbuchstabe
- T für Type
- K für Key
- V für Value
- E für Element
- ....



ohne Generics



```
1 public class
  ↳ StringAndDoubleHolder {
2     public String valueA;
3     public Double valueB;
4 }
5 public class
  ↳ IntegerAndShortHolder {
6     public Integer valueA;
7     public Short valueB;
8 }
```

# Generics-Grundlagen

## Mehrere Typparameter



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



ohne Generics



```
1 public class
  ↳ StringAndDoubleHolder {
2     public String valueA;
3     public Double valueB;
4 }
5 public class
  ↳ IntegerAndShortHolder {
6     public Integer valueA;
7     public Short valueB;
8 }
```



mit Generics



```
1 public class Tuple<A, B> {
2     public A valueA;
3     public B valueB;
4 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

**Diamond operator**

Generische Methoden

Eingeschränkte Typparameter

Erben von generischen Klassen

Erstellen von generischen Arrays

Wildcards

Type Erasure



## Diamond Operator



```
1 ArrayList<String> myList1 = new ArrayList<String>();  
2 ArrayList<String> myList2 = new ArrayList<>(); // String wird impliziert  
3 List<String> myList3 = new ArrayList<>();
```



## Diamond Operator



```
1 ArrayList<String> myList1 = new ArrayList<String>();
2 ArrayList<String> myList2 = new ArrayList<>(); // String wird impliziert
3 List<String> myList3 = new ArrayList<>();
4
5 Tuple<String, Integer> myTuple1 = new Tuple<>();
6 Tuple<Tuple<String, Integer>, Tuple<List<String>, Integer>> myTuple1
7     = new Tuple<>();
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

Diamond operator

**Generische Methoden**

Eingeschränkte Typparameter

Erben von generischen Klassen

Erstellen von generischen Arrays

Wildcards

Type Erasure



```
1 public List<Double> nonGenericFilter(List<Double> list,  
   ↪ Predicate<Double> predicate) {  
2     // ....  
3 }  
4 public List<Boolean> nonGenericFilter(List<Boolean> list,  
   ↪ Predicate<Boolean> predicate) {  
5     // ....  
6 }
```





```
1 public List<Double> nonGenericFilter(List<Double> list,  
   ↪ Predicate<Double> predicate) {  
2     // ....  
3 }  
4 public List<Boolean> nonGenericFilter(List<Boolean> list,  
   ↪ Predicate<Boolean> predicate) {  
5     // ....  
6 }
```

```
1 public <T> List<T> filter(List<T> list, Predicate<T> predicate) {  
2     // ....  
3 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

Diamond operator

Generische Methoden

**Eingeschränkte Typparameter**

- extends Klassen**

- extends Interfaces**

- extends Klassen und Interfaces**

Erben von generischen Klassen

Erstellen von generischen Arrays

Wildcards

Type Erasure

# Eingeschränkte Typparameter

extends Klassen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Holder<T> {  
2     public T value;  
3     public void test() {  
4         int i = // ...?  
5     }  
6 }
```

# Eingeschränkte Typparameter

## extends Klassen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Holder<T extends Number> {  
2     public T value;  
3     public void test() {  
4         int i = value.intValue();  
5     }  
6 }
```

# Eingeschränkte Typparameter

## extends Klassen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Holder<T extends Number> {  
2     public T value;  
3     public void test() {  
4         int i = value.intValue();  
5     }  
6 }
```

```
1 Holder<Integer> holder1 = new Holder<Integer>();  
2 Holder<Boolean> holder2 = new Holder<Boolean>(); // !!
```

# Eingeschränkte Typparameter

extends Interfaces



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Holder<T extends Inf1 & Inf2 & Inf3> {  
2     public T value;  
3     // ...  
4 }
```

# Eingeschränkte Typparameter

extends Klassen und Interfaces



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Holder<T extends Number & Inf1 & Inf2 & Inf3> {  
2     public T value;  
3     // ...  
4 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

Diamond operator

Generische Methoden

Eingeschränkte Typparameter

**Erben von generischen Klassen**

Erstellen von generischen Arrays

Wildcards

Type Erasure





```
1 public class MyGenericClass<T> {  
2     // ...  
3 }
```

# Erben von generischen Klassen

```
1 public class MyGenericClass<T> {  
2     // ...  
3 }
```

```
1 public class Sub<Q> extends MyGenericClass<Q> {  
2     // ...  
3 }
```

# Erben von generischen Klassen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class MyGenericClass<T> {  
2     // ...  
3 }
```

```
1 public class Sub<Q> extends MyGenericClass<Q> {  
2     // ...  
3 }
```

```
1 public class IntegerSub extends MyGenericClass<Integer> {  
2     // ...  
3 }
```



```
1 public class MyGenericClass<T extends Integer> {  
2     // ...  
3 }
```



```
1 public class MyGenericClass<T extends Integer> {  
2     // ...  
3 }
```

```
1 public class IntegerSub extends MyGenericClass<Integer> {  
2     // ...  
3 }
```



```
1 public class MyGenericClass<T extends Integer> {  
2     // ...  
3 }
```

```
1 public class IntegerSub extends MyGenericClass<Integer> {  
2     // ...  
3 }
```

```
1 public class BooleanSub extends MyGenericClass<Boolean> { // !!  
2     // ...  
3 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

Diamond operator

Generische Methoden

Eingeschränkte Typparameter

Erben von generischen Klassen

**Erstellen von generischen Arrays**

Wildcards

Type Erasure

# Erstellen von generischen Arrays



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class MyGenericClass<T> {  
2     public T[] createMyArray() {  
3         T[] testArray = (T[]) new Object[25];  
4         return testArray;  
5     }  
6 }
```



# Erstellen von generischen Arrays



```
1 public class MyGenericClass<T> {  
2     public T[] createMyArray() {  
3         T[] testArray = (T[]) new Object[25];  
4         return testArray;  
5     }  
6 }
```

```
1 public <Q> Q[] createMyArray() {  
2     Q[] testArray = (Q[]) new Object[25];  
3     return testArray;  
4 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

Diamond operator

Generische Methoden

Eingeschränkte Typparameter

Erben von generischen Klassen

Erstellen von generischen Arrays

**Wildcards**

Grundlagen

Wann werden Wildcards verwendet?

? Wildcards mit super

Kovarianz - Parameter

Kovarianz - Rückgabetyp

**Kovarianz - Arrays**

Invarianz



```
1 public <T> void printList(List<T> list) {  
2     for (T item : list) {  
3         System.out.println(item);  
4     }  
5 }
```



```
1 public <T> void printList(List<T> list) {  
2     for (T item : list) {  
3         System.out.println(item);  
4     }  
5 }
```

```
1 public void printList(List<?> list) {  
2     for (Object item : list) {  
3         System.out.println(item);  
4     }  
5 }
```



```
1 public <T extends Number> void printList(List<T> list) {  
2     for (T item : list) {  
3         System.out.println(item.intValue() + 1);  
4     }  
5 }
```



```
1 public <T extends Number> void printList(List<T> list) {  
2     for (T item : list) {  
3         System.out.println(item.intValue() + 1);  
4     }  
5 }
```

```
1 public void printList(List<? extends Number> list) {  
2     for (Number item : list) {  
3         System.out.println(item.intValue() + 1);  
4     }  
5 }
```

# Wildcards

## Wann werden Wildcards verwendet?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void nonGenMethod(Holder<? extends Number> a, Holder<?> b) {  
2     // ...  
3 }
```

# Wildcards

## Wann werden Wildcards verwendet?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void nonGenMethod(Holder<? extends Number> a, Holder<?> b) {  
2     // ...  
3 }
```

```
1 public <T extends Number> void genMethodA(Holder<T> a, Holder<T> b) {  
2     // ...  
3 }
```



# Wildcards

## Wann werden Wildcards verwendet?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void nonGenMethod(Holder<? extends Number> a, Holder<?> b) {  
2     // ...  
3 }
```

```
1 public <T extends Number> void genMethodA(Holder<T> a, Holder<T> b) {  
2     // ...  
3 }
```

```
1 public <T extends Number, Q> void genMethodB(Holder<T> a, Holder<Q> b) {  
2     // ...  
3 }
```

# Wildcards

## ? Wildcards mit super



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void nonGenMethod(Predicate<? super Number> a) {  
2     a.intValue(); // !! Nicht erlaubt  
3 }
```

# Wildcards

## ? Wildcards mit super



```
1 public void nonGenMethod(Predicate<? super Number> a) {  
2     a.intValue(); // !! Nicht erlaubt  
3 }
```

```
1 Predicate<? super Number> a = (Object x) -> true;  
2 Predicate<? super Number> b = (Serializable x) -> true;  
3 Predicate<? super Number> c = (Number x) -> true;
```



### Generic Variance

- ? extends T - Kovarianz (Lesender Zugriff)
- ? super T - Kontravarianz (Schreibender Zugriff)
- T - Invarianz (Beides)



```
1 public class A {}  
2 public class B extends A {}
```



```
1 public class A {}  
2 public class B extends A {}
```

```
1 public void foo(A a) { ... }
```



```
1 public class A {}  
2 public class B extends A {}
```

```
1 public void foo(A a) { ... }
```

```
1 foo(new A());  
2 foo(new B()); // parameter covariance
```



```
1 public class A {}  
2 public class B extends A {}
```





```
1 public class A {}  
2 public class B extends A {}
```

```
1 public class Baz {  
2     public A foo() {...}  
3 }  
4 public class SpecialBaz extends Baz {  
5     @Override  
6     public B foo() {...} // return type covariance  
7 }
```



```
1 public class A {}  
2 public class B extends A {}
```



```
1 public class A {}  
2 public class B extends A {}
```

```
1 A[] myArray = new B[2]; // array covariance
```



```
1 public class A {}  
2 public class B extends A {}
```

```
1 A[] myArray = new B[2]; // array covariance
```

```
1 A[] myArray = new A[2];  
2 myArray[0] = new B(); // array covariance
```



```
1 public class A {}  
2 public class B extends A {}  
3 public class C extends A {}
```



```
1 public class A {}  
2 public class B extends A {}  
3 public class C extends A {}
```

```
1 A[] myArray = new B[2];  
2 myArray[0] = new B();
```



```
1 public class A {}  
2 public class B extends A {}  
3 public class C extends A {}
```

```
1 A[] myArray = new B[2];  
2 myArray[0] = new B();
```

```
1 A[] myArray = new B[2];  
2 myArray[0] = new C(); // !! array covariance
```



```
1 public class A {}  
2 public class B extends A {}
```





```
1 public class A {}  
2 public class B extends A {}
```

```
1 List<B> myList = new ArrayList<B>();  
2 List<A> otherList = myList; // !!  
3  
4 List<? extends A> list = myList;
```



```
1 public class A {}  
2 public class B extends A {}
```

```
1 List<B> myList = new ArrayList<B>();  
2 List<A> otherList = myList; // !!  
3  
4 List<? extends A> list = myList;  
5 list.add(new A()); // !! compile error  
6 list.add(new B()); // !! compile error
```



```
1 public void readFromList(List<? extends Number> list) {  
2     // z.B: List<Double>, List<Integer>, List<Number>  
3     Number a = list.get(0);  
4 }  
5 public void readFromList(List<? super Number> list) {  
6     // z.B: List<Number>, List<Object>  
7     Number a = list.get(0); // !!  
8 }
```

# Wildcards

Wann super, wann extends?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void writeToList(List<? extends Number> list) {  
2     // z.B: List<Double>, List<Integer>, List<Number>  
3     list.add(Double.valueOf(2.0)); // !!  
4 }  
5 public void readFromList(List<? super Number> list) {  
6     // z.B: List<Number>, List<Object>  
7     list.add(Double.valueOf(2.0));  
8 }
```

# Wildcards

Wann super, wann extends?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void both(List<? extends Number> list) {  
2     // z.B: List<Double>, List<Integer>, List<Number>  
3     Number a = list.get(0);  
4     list.add(Double.valueOf(2.0)); // !!  
5 }  
6 public void readFromList(List<? super Number> list) {  
7     // z.B: List<Number>, List<Object>  
8     Number a = list.get(0); // !!  
9     list.add(Double.valueOf(2.0));  
10 }
```

# Wildcards

Wann super, wann extends?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void both(List<Double> list) {  
2     // nur List<Double>  
3     Number a = list.get(0);  
4     list.add(Double.valueOf(2.0));  
5 }
```



- Supplier: In-Parameter
- Consumer: Out-Parameter
- Function: Beides

# Wildcards

## Wann super, wann extends? Beispiele



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public interface Supplier<T> {  
2     T get();  
3 }
```





? **super** T (Kontravarianz)

```
1 public void foo(Supplier<? super T> supplier) {  
2     T a = supplier.get(); // !!  
3 }
```



### T (Invarianz)

```
1 public void foo(List<T> list, Supplier<T> supplier) {  
2     list.add(supplier.get());  
3 }  
4 public void bar() {  
5     List<Number> list = new ArrayList<>();  
6     Supplier<Integer> supplier = () -> 1;  
7     foo(list, supplier); // !!  
8 }
```



? **extends** T (Kovarianz)

```
1 public void foo(List<T> list, Supplier<? extends T> supplier) {  
2     list.add(supplier.get());  
3 }  
4 public void bar() {  
5     List<Number> list = new ArrayList<>();  
6     Supplier<Integer> supplier = () -> 1;  
7     foo(list, supplier); // geht jetzt  
8 }
```

# Wildcards

## Wann super, wann extends? Beispiele



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public interface Consumer<T> {  
2     void accept(T t);  
3 }
```



? **extends** T (Kovarianz)

```
1 public void foo(Consumer<? extends Number> consumer) {  
2     consumer.accept(42); // !!  
3 }
```



### T (Invarianz)

```
1 public void foo(List<T> list, Consumer<T> consumer) {  
2     consumer.accept(list.get(0));  
3 }  
4 public void bar() {  
5     List<Integer> list = new ArrayList<>();  
6     Consumer<Number> consumer = n -> System.out.println(n.intValue());  
7     foo(list, consumer); // !!  
8 }
```



? **super** T (Kontravarianz)

```
1 public void foo(List<T> list, Consumer<? super T> consumer) {  
2     consumer.accept(list.get(0));  
3 }  
4 public void bar() {  
5     List<Integer> list = new ArrayList<>();  
6     Consumer<Number> consumer = n -> System.out.println(n.intValue());  
7     foo(list, consumer); // geht jetzt  
8 }
```



```
1 public interface Function<T, R> {  
2     R apply(T t);  
3 }
```





- `Supplier` → `Supplier<? extends T>`
- `Consumer` → `Consumer<? super T>`
- `Function` → `Function<? super T, ? extends R>`

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Mehrdimensionale Arrays

Generics-Grundlagen

Diamond operator

Generische Methoden

Eingeschränkte Typparameter

Erben von generischen Klassen

Erstellen von generischen Arrays

Wildcards

**Type Erasure**



- Typparameter sind zur Laufzeit nicht bekannt!
- Im Prinzip nur Schutz vor Fehlern beim Kompilieren



```
1 public class Test<T extends Number> {  
2     T val;  
3     public void test(T p) {  
4     }  
5 }
```



```
1 public class Test<T extends Number> {  
2     T val;  
3     public void test(T p) {  
4     }  
5 }
```

```
1 public class Test<Number> {  
2     Number val;  
3     public void test(Number p) {  
4     }  
5 }
```



```
1 public class Test<T> {  
2     T val;  
3     public void test(T p) {  
4     }  
5 }
```



```
1 public class Test<T> {  
2     T val;  
3     public void test(T p) {  
4     }  
5 }
```

```
1 public class Test<Object> {  
2     Object val;  
3     public void test(Object p) {  
4     }  
5 }
```



---

# Live-Coding!