Generics<T>

Introduction & Overview

29.03.22

Inden Michael

ad cubum
think.insurance

Agenda

- Motivation und Intro
- EXERCISES
- Varianzformen
- Generics und Collections
- EXERCISES
- Besonderheiten
- EXERCISES

Motivation und Intro

Motivation und Intro

- Warum Generics? => Typsichere Klassen bzw. Container bereitstellen können
- Bedeutung nicht nur im Namen codieren, sondern durch Compiler prüfen können

```
private static void simpleDefintionAndUsage() {
    List stringList = new ArrayList();
    stringList.add("Hallo");
    String str = (String) stringList.get(0);

    List<String> stringList2 = new ArrayList<String>();
    stringList2.add("Hallo");
    String str2 = stringList2.get(0);
}
```

Motivation und Intro

- Probleme ***heterogener*** Container

```
private static void misuseNoGenerics() {
    List stringList = new ArrayList();
    stringList.add("Hallo");
    stringList.add(-1234);
    String str = (String) stringList.get(1); // ← Exception zur Laufzeit
}


private static void misuseWithGenerics() {
    List<String> stringList2 = new ArrayList<String>();
    stringList2.add("Hallo");
    stringList2.add(-1234); // ← Kompilierfehler
    String str2 = stringList2.get(1);
}
```

Motivation und Intro

- Abhilfe durch ***homogene*** Container mit Generics

```
public static void main(final String[] args)
{
    final List<Person> personList = new ArrayList<>();
    personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
    personList.add(new Person("Moritz", LocalDate.now(), "Musterstadt"));
    // personList.add(new Dog(''Sarah vom Auetal''));  // Compile-Error

    for (int i = 0; i < personList.size(); i++)
    {
        final Person person = personList.get(i);
        System.out.println(person.getName() + " aus " + person.getCity());
    }
}
```

Motivation und Intro

- Eigene generische Klassen

```
public class SimpleGenericClass<T> {

    private T data;

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
```

Motivation und Intro

- Eigene generische Klassen mit Typeinschränkung

```
public class SimpleGenericClass<T extends BaseType & Interface1 & Interface2> {

    private T data;

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
```

- T muss die nach extends angegebene Basisklasse bzw. den Basistyp BaseType besitzen (oder selbst vom Typ BaseType sein).
- Optional können mit & Interfaces spezifiziert werden: T muss im Beispiel dann auch die Interfaces Interface1 und Interface2 implementieren. Diese Bedingung ist auch erfüllt, wenn eine beliebige Basisklasse von T dies tut.

Motivation und Intro

- Eigene generische Methoden

```
static <T> int findElemFromPos(List<T> values, T elemToFind) {
    return findElemFromPos(values, elemToFind, 0);
}

static <T> int findElemFromPos(List<T> values, T elemToFind, int startPos)
{
    for (int i = startPos; i < values.size(); i++) {
        if (values.get(i).equals(elemToFind)) {
            return i;
        }
    }
    return -1;
}
```

Generische Methoden mit Typanforderungen

```java
public static <T> T maxOf3(T x, T y, T z) {
    T max = x;

    if (y.compareTo(max) > 0) {
        max = y;
    }

    if (z.compareTo(max) > 0) {
        max = z;
    }

    return max;
}
```

The method compareTo(T) is undefined for the type T

=>

```java
maxOf3(1, 7, 2)
maxOf3(6.6, 8.8, 7.7)
maxOf3("Java", "Python", "C#")
```

Generische Methoden mit Typanforderungen

```java
public static <T extends Comparable<T>> T maxOf3(T x, T y, T z) {
    T max = x;

    if (y.compareTo(max) > 0) {
        max = y;
    }

    if (z.compareTo(max) > 0) {
        max = z;
    }

    return max;
}
```

=>

```java
maxOf3(1, 7, 2)
maxOf3(6.6, 8.8, 7.7)
maxOf3("Java", "Python", "C#")
```

Generische Methoden für Var Args

```
public static <T extends Comparable<T>> T maxOf(T... values) {
    T max = values[0];

    for (int i = 1; i < values.length; i++)
    {
        if (values[i].compareTo(max) > 0) {
            max = values[i];
        }
    }

    return max;
}
```

Generische Methoden für Listen

```java
public static <T extends Comparable<T>> T maxOf(List<T> values) {
    T max = values.get(0);

    for (int i = 1; i < values.size(); i++)
    {
        if (values.get(i).compareTo(max) > 0) {
            max = values.get(i);
        }
    }

    return max;
}
```


examples/GenericeMethod_MaxOf3.java

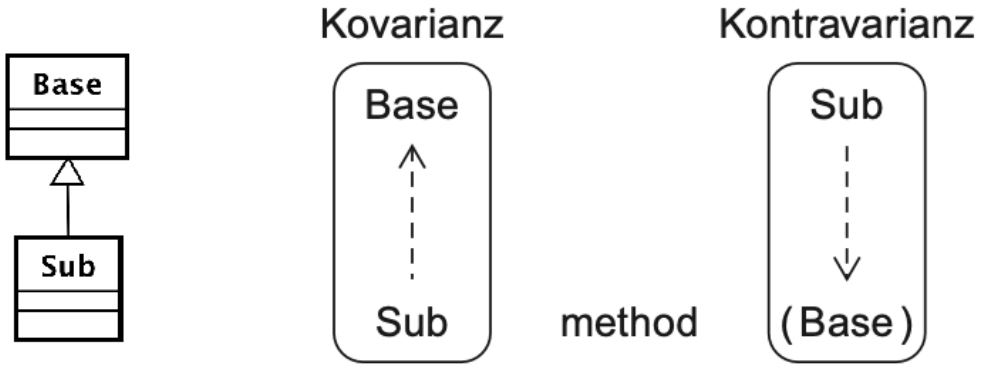
EXERCISES 1– 3

Varianzformen

Habt ihr schon mal was von Varianz gehört?

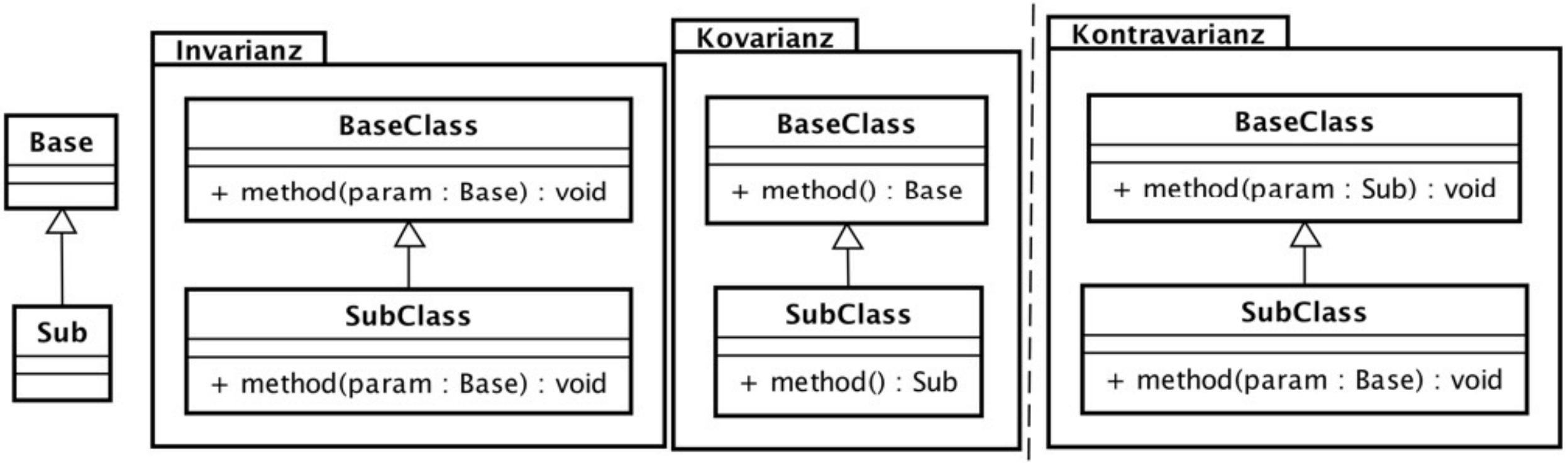
Varianzformen

- Laut »is-a«-Beziehung und Substitutionsprinzip => an allen Stellen in einem Programm, an denen ein Basistyp verwendet wird, es auch möglich sein sollte, jeden Subtyp zu verwenden.



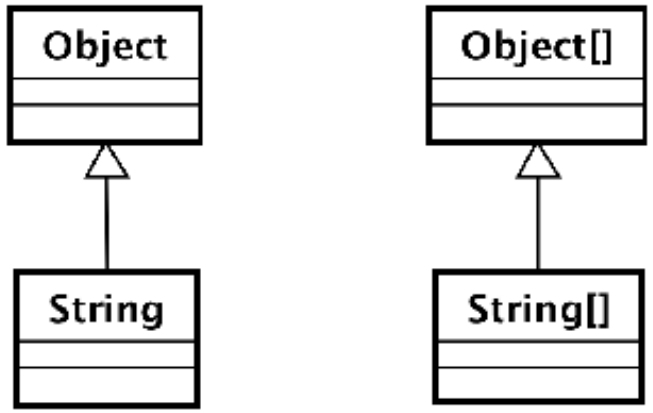
- Für Methodenparameter gilt: Ein **Eingabetyp** darf **spezieller** sein als der von der Methode in der Signatur geforderte Eingabetyp, da in der Methode lediglich die Attribute und Methoden der Basisklasse zugreifbar sind und diese auch in der Subklasse vorhanden sind.
- Auch für **Rückgabetypen** darf ein **speziellerer** Typ zurückgeliefert werden, da ein verwendender Aufrufer nur die Attribute und Methoden der Basisklasse kennt.
- Varianz beschreibt die beiden gezeigten Typabweichungen Ko- und Kontravarianz.

Varianzformen



Kovarianz bei Arrays

- Die Kovarianz besagt in diesem Fall das Folgende: Da die Klasse String ein Subtyp von Object ist, gilt das auch für Arrays dieser Typen: Ein String[] stellt in Java somit einen Subtyp von Object[] dar.



- Scheint absolut logisch!
- Haben wir immer einfach so ohne Hinterfragen genutzt:

```
final Object[] messages = new String[] { "Arrays", "sind", "kovariant" };
```

ad cubum

Habt ihr dabei schon mal eine ArrayStoreException bekommen? Was ist das überhaupt?

Kovarianz bei Arrays

- Die Kovarianz ermöglicht es, beliebige, nicht primitive Array-Typen an Object[] zuzuweisen:

```
Object[] messages = new String[] { "Arrays", "sind", "kovariant" };
```

- zwischenzeitlich auch auf ein Integer[] und später wieder auf ein String[] verweisen.

```
messages = new Integer[] { 1, 2, 3 };

messages = new String[] { "Back", "to", "Strings" };

// Typfehler, weil Message nicht kompatibel zu String ist
messages[1] = new Message("Typfehler durch Kovarianz!");
```

```
Exception in thread "main" java.lang.ArrayStoreException:
examples.ArrayCovarianceExample$Message
```

- **Typsicherheit nicht zur Kompilierzeit sichergestellt => erfordert eine Prüfung zur Laufzeit!**

Generics und Collections

Generics und Collections

- Als Generics mit JDK 5 eingeführt werden sollten, stand man vor der Herausforderung, dies kompatibel zur JVM und den bisherigen Klassen des JDKs zu realisieren.
- Dies war nur durch einen Trick möglich: Generics sind mithilfe der sogenannten ***Type Erasure*** realisiert. Das bedeutet, dass die im Sourcecode typisierten Klassen durch den Compiler auf untypisierte Klassen bzw. die Parameter auf den allgemeinsten Typ Object abgebildet werden:

```
final List<Person> personList = new ArrayList<>();
personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
final Person firstPerson = personList.get(0);

=>

final List personList = new ArrayList();
personList.add(new Person("Max", LocalDate.now(), "Musterstadt"));
final Person firstPerson = (Person)personList.get(0);
```

Generics und Collections

- Generics und Collections kann man ohne viel Nachdenken folgendermassen verwenden:

```
final List<String> names = new ArrayList<>();
final Set<BaseFigure> figures = new HashSet<>();
```

- Für Arrays kann man wegen Kovarianz folgendes schrieben:

```
final Object[] names = new String[10];
final BaseFigure[] figures = new Circle[10];
```

- Was passiert bei Generics?

```
// Achtung: Kompilierfehler
final List<Object> names = new ArrayList<String>();
final Set<BaseFigure> figures = new HashSet<Circle>();
```

Generics und Collections

- Generics und Collections sind per default **INVARIANT**

```
final Set<BaseFigure> figures = new HashSet<BaseFigure>();
//      ^       ^                          ^           ^
//      |       |_____ Typparameter invariant _____|
//      |                                    |
//      |_______ Typen kovariant _______|
```

Mit ***Invarianz*** ist für Generics gemeint, dass ***die Typparameter bei Deklaration und Definition exakt übereinstimmen.*** Diese Forderung ist sehr restriktiv. ***Kovarianz*** bedeutet, dass die Typen der Vererbungshierarchie folgen. Dies ist hier für das Interface `Set<E>` und die konkrete Realisierung `HashSet<E>` gegeben.

Generics und Collections

- Generics und Collections sind per default **INVARIANT**

```
// Compile-Error
final Set<BaseFigure> figures = new HashSet<RectFigure>();
//            ^                                   ^
//            |____ Typparameter kovariant _____|
```

Generics und Collections

```
// Compile-Error
final List<Object> names = new ArrayList<String>();
//             ^                              ^
//             |__ Typparameter kovariant ___|

names.add(new RectFigure());  // Typinkompatibilität
```

Ebenso wie für Arrays sollte dies dann zur Laufzeit zu einem Fehler führen, wenn man einer `List<Object>` mal eine `List<String>` oder `List<BaseFigure>` zuweist und Elemente hinzufügt. Um den Typverstoß aber erkennen zu können, müsste man für die Containerklassen des JDKs analog zu Arrays zur Laufzeit Typinformationen zu den gespeicherten Elementen besitzen. Nun wurde aber beim Entwurf von Generics die Entscheidung getroffen, Generics mithilfe von Type Erasure (vgl. Abschnitt 3.7.2) umzusetzen. Aufgrund dessen ist eine Typprüfung zur Laufzeit keine Option, da nach dem Kompilieren keine Typinformationen mehr vorhanden sind, die ausgewertet werden könnten. Wäre obige kovariante Definition also erlaubt, so könnte der Compiler hier nicht mehr sicherstellen, dass zur Laufzeit keine Typinkompatibilitäten auftreten. Daher führt die gezeigte Definition bereits beim Kompilieren zu einem Fehler.

Generics und Collections

Problemlösung Um die mitunter wünschenswerte Kovarianz für Typparameter bei Generics zu ermöglichen, existiert eine spezielle Notation. Allerdings darf aufgrund der Type Erasure und der fehlenden Möglichkeit einer Typprüfung zur Laufzeit nur eine »sichere Form« der Kovarianz unterstützt werden, bei der garantiert wird, dass der verwendete Typ alle Operationen des Basistyps implementiert. Gleiches gilt für Kontravarianz. Für beide kommt eine sogenannte Wildcard zum Einsatz. Die Wildcard '`?`' erlaubt beliebige Typen und kann zur Kennzeichnung von Ko- und Kontravarianz wie folgt genutzt werden:

1. **Kovarianz** – Die Wildcard '`? extends` **`basetype`**' wird ***Upper Type Bound*** genannt und ermöglicht ***Kovarianz***. Es dürfen dadurch generische Klassen verwendet werden, die Subtypen von `basetype` als Typparameter nutzen.[22]
2. **Kontravarianz** – Die Wildcard '`? super` **`subtype`**' heißt ***Lower Type Bound*** und ermöglicht ***Kontravarianz***. Dadurch sind alle diejenigen generischen Klassen erlaubt, die als Typparameter einen Basistyp von `subtype` besitzen. Es werden für die Typparameter also *alle* Basistypen akzeptiert, jedoch *keine* Subtypen mehr.

Generics und Collections

- **Upper Type Bound => eine generische Methode für mehrere Typen mit gemeinsamer Basis bereitstellen**

```
final List<? extends BaseFigure> figureList1 = new ArrayList<CircleFigure>();
final List<? extends BaseFigure> figureList2 = new ArrayList<RectFigure>();
```

```
private static Number sum(List<? extends Number> numbers) {
    double s = 0.0;

    for (Number n : numbers) {
        s += n.doubleValue();
    }

    return s;
}
```

Generics und Collections

- **Upper Type Bound => eine generische Methode für mehrere Typen mit gemeinsamer Basis bereitstellen**

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);
System.out.println(sum(ints));

List<Double> doubles = List.of(1.5d, 2d, 3d);
System.out.println(sum(doubles));

List<String> strings = List.of("1", "2");
System.out.println(sum(strings));
```

The method sum(List<? extends Number>) in the type SimpleGenericMethodWithExtends is not applicable for the arguments (List<String>)

Generics und Collections

- **Upper Type Bound => eine generische Methode für mehrere Typen mit gemeinsamer Basis bereitstellen**

```
private static Number specialsum(List<? extends Number> numbers) {
    double s = 0.0;
    for (Number n : numbers)
        s += n.doubleValue();

    // add VAT
    numbers.add(s * 0.19d);
    return s;
}
```

- **Zugriff erlaubt**
- **Einfügen von Werten NICHT erlaubt, wegen Typunsicherheit wie bei Arrays**

Kontravarianz – Lower Type Bound

Die Forderung nach Kovarianz ist relativ natürlich und basiert auf dem Substitutionsprinzip. Der Einsatz von Kontravarianz ist weniger eingängig, weil diese eine Kompatibilität entgegengesetzt zur Vererbungshierarchie der Typparameter ermöglicht. Kontravarianz bei Generics wird durch die Notation '`? super subtype`' ausgedrückt:

```
final List<? super BaseFigure> figureList = new ArrayList<BaseFigure>();
```

Im Speziellen ist sogar folgende Zuweisung möglich:

```
final List<? super BaseFigure> figureList = new ArrayList<Object>();
```

Generics und Collections

- **Lower Type Bound**

```
public static void main(String[] args) {
    List<Number> ints = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
    System.out.println(addVAT(ints, 15));

    List<Number> doubles = new ArrayList<>(List.of(1.5d, 2d, 3d));
    System.out.println(addVAT(doubles, 6.5d));
}

private static List<?> addVAT(List<? super Number> numbers, Number sum) {
    // add VAT
    numbers.add(sum.doubleValue() * 0.19d);
    return numbers;
}
```

- **Zugriff NICHT erlaubt**
- **Einfügen von Werten erlaubt**

Invarianz

Bekanntermaßen sind bei der Invarianz für Generics die Typparameter bei Deklaration und Definition gleich. Im folgenden Beispiel ist dies für eine `ArrayList` von `BaseFigure`-Objekten gezeigt:

```
final List<BaseFigure> graphicObjects = new ArrayList<BaseFigure>();
```

Eine auf diese Weise definierte `ArrayList<BaseFigure>` lässt sich so nutzen, als ob die Signaturen der Methoden tatsächlich für `BaseFigure`-Objekte definiert wären:

Beim Auslesen mit `get(int)` liefert sie ein Objekt vom Typ `BaseFigure`. Beim Hinzufügen lässt sich mit `add(BaseFigure)` ein Element vom Typ `BaseFigure` bzw. sogar Subtypen davon speichern:

```
final BaseFigure baseFigure = graphicObjects.get(index);
graphicObjects.add(new RectFigure());
```

Zugriff erlaubt
Einfügen von Werten erlaubt

Generics und Collections

- **Schreiben wir doch mal eine copy()-Funktionalität**

```
public static void copy(final List<BaseFigure> src, final List<BaseFigure> dest) {
    for (final BaseFigure figure : src) {
        dest.add(figure);
    }
}
```

Was sind mögliche Probleme?

- **Problem**

Problem Invarianz Das erste Problem der gezeigten Realisierung besteht in der invarianten Definition der Typparameter von Quelle und Ziel:

```
final List<BaseFigure> src = new ArrayList<BaseFigure>();
// Füllen der Liste ...
final List<BaseFigure> dest = new ArrayList<BaseFigure>();

FigureUtilities.copy(src, dest);
```

Dadurch sind wir darauf beschränkt, beim Kopieren jeweils nur Listen mit identischem Typparameter zu nutzen. Das ist jedoch störend, wenn man Subtypbeziehungen und Polymorphie nutzt.

```
final List<CircleFigure> src = new ArrayList<CircleFigure>();
// Füllen der Liste ...
final List<BaseFigure> dest = new ArrayList<BaseFigure>();

// Compile-Error: The method copy(List<BaseFigure>, List<BaseFigure>)
// in the type FigureUtilities is not applicable for the arguments
// (List<CircleFigure>, List<BaseFigure>)
FigureUtilities.copy(src, dest);
```

- **Lösung**

Motivation für Kovarianz für den Parameter der Quelle Als Abhilfe nutzen wir eine kovariante Definition folgendermaßen:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

Damit lässt sich die eben noch an einem Kompilierfehler scheiternde Kopieraktion von einer `ArrayList<CircleFigure>` in eine `ArrayList<BaseFigure>` durchführen.

Generics und Collections

Varianzform	Lesezugriff	Schreibzugriff
Kontravarianz	- (nur `Object`)	✓
Kovarianz	✓	- (nur `null`)
Invarianz	✓	✓

EXERCISES 4 – 7

Besonderheiten

Besonderheiten – Was geht mit Generics nicht?

```
public class StaticAttributeExample<T>
{
   private static T member; // This is not allowed
}
```

```
public class DynamicArrayCreation<T> {
    public DynamicArrayCreation() {
        new T(); // This is not allowed
    }
}
```

Woran könnte das liegen?

Besonderheiten – Was geht mit Generics nicht?

ad cubum

Type Erasure !!!!!!!

Besonderheiten – Was geht mit Generics nicht?

```
static class Matrix<T> {
    private T[][] values;
    private int rows;
    private int cols;

    public Matrix(int r, int c) {
        rows = r;
        cols = c;
        // values = new T[rows][cols]; // THIS WILL NOT COMPILE
    }

    public void set(int r, int c, T v) {
        values[r][c] = v;
    }

    public T get(int r, int c) {
         return values[r][c];
    }
}
```

Besonderheiten – Was geht mit Generics nicht?

```
static class Matrix<T extends Object> {
    private Object[][] values;
    private int rows;
    private int cols;

    public Matrix(int r, int c) {
        rows = r;
        cols = c;
        values = new Object[rows][cols]; // WARNING
    }

    public void set(int r, int c, T v) {
        values[r][c] = v;
    }

    public T get(int r, int c) {
         return (T)values[r][c];
    }
}
```

Besonderheiten List, List<?> und List<Object>

`List` vs. `List<Object>` Etwas vereinfachend kann man sagen, dass die Deklarationen `List` und `List<Object>` nahezu gleich sind. Derart definiert können Elemente beliebigen Typs gespeichert werden. Der Unterschied besteht lediglich darin, welche Zuweisungen an die so definierten Referenzvariablen vorgenommen werden können: Eine Deklaration `List<Object>` erlaubt tatsächlich nur, dass exakt so definierte Listen zugewiesen werden. Das folgt aufgrund der Invarianz:

```
// beim Auslesen nicht typsicher
final List plainList = new ArrayList();
plainList.add(new Integer(4711));
plainList.add("Test");
final Integer sum = plainList.get(0) + // Laufzeitfehler, weil
                    plainList.get(1);  // Integer und String enthalten sind

// ähnlich wie List, total generisch
final List<Object> objectList = new ArrayList<Object>();
// final List<Object> objectList = new ArrayList<String>(); // Compile-Error
objectList.add("Test");
objectList.add(new Integer(4711));
```

Besonderheiten List, List<?> und List<Object>

`List<Object>` vs. `List<?>` Im Gegensatz zu den gerade betrachteten Deklarationen bestehen zwischen den Deklarationen `List<Object>` und `List<?>` sehr große Unterschiede. Wie eben erwähnt, können in einer `List<Object>` Elemente beliebigen Typs gespeichert werden, aufgrund der Invarianz jedoch nur Listen zugewiesen werden, die auch exakt den generischen Typ `Object` nutzen.

Für die `List<?>` gilt das Gegenteil: Die Wildcard '`?`' erlaubt beliebige Typen und somit können Listen mit beliebigen Typparametern zugewiesen werden:

```
final List<?> anyTypeList1 = new ArrayList<Person>();
final List<?> anyTypeList2 = new ArrayList<CircleFigure>();

// typsichere Liste, bei der der Typ unbekannt ist
final List<?> anyTypeList = new ArrayList<String>();

// Compile-Error: The method add(capture#1-of ?) in the type
// List<capture#1-of ?> is not applicable for the arguments (String)/(Object)
// anyTypeList.add("Test");
// anyTypeList.add("Object");

anyTypeList.add(null);     // erlaubt
```

Besonderheiten List, List<?> und List<Object>

Zusammenfassung Fassen wir alles noch mal zusammen: Wie eingangs erwähnt, verhält sich eine `List<Object>` nahezu wie der nicht typisierte Raw Type `List` und definiert eine ***heterogene Liste***. Allerdings kann einer `List<Object>` aufgrund der Forderung nach Invarianz *keine* `List<String>` zugewiesen werden; es können der `List<Object>` aber durchaus Stringobjekte hinzugefügt werden.

Mit einer `List<?>` wird eine ***homogene Liste*** beschrieben: Es kann zwar eine `List<String>` oder `List<Person>` zugewiesen werden; es können aber *keine* Stringobjekte bzw. `Person`-Objekte hinzugefügt werden. Dies klingt nach einer starken Einschränkung. Trotzdem ist `List<?>` aber für Utility-Methoden nützlich, die nur auf der Listenfunktionalität unabhängig vom Typ arbeiten sollen, etwa `size(List<?>)`.

Besonderheiten Typhilfe nötig

Spezielle Syntax bei Generics

Wir haben mit Ko- und Kontravarianz und der ?-Notation schon eine recht kryptische Syntax bei Generics kennengelernt. Leider gibt es sogar noch eine Steigerung. Diese kommt dann zum Einsatz, wenn man Methoden mit generischen Parametern aufrufen möchte, etwa folgendermaßen:

```
public void print(final String title,
                  final List<TimeSeriesData> timeSeriesData);
```

Diese Signatur sieht harmlos aus, aber was passiert, wenn wir eine leere Liste als Parameter übergeben wollen? Dazu nutzen wir `Collections.emptyList()` wie folgt:

```
print("title", Collections.emptyList());
```

Besonderheiten Typhilfe nötig

ad cubum

Spezielle Syntax bei Generics

Wir haben mit Ko- und Kontravarianz und der ?-Notation schon eine recht kryptische Syntax bei Generics kennengelernt. Leider gibt es sogar noch eine Steigerung. Diese kommt dann zum Einsatz, wenn man Methoden mit generischen Parametern aufrufen möchte, etwa folgendermaßen:

```
public void print(final String title,
                  final List<TimeSeriesData> timeSeriesData);
```

Diese Signatur sieht harmlos aus, aber was passiert, wenn wir eine leere Liste als Parameter übergeben wollen? Dazu nutzen wir `Collections.emptyList()` wie folgt:

```
print("title", Collections.emptyList());
```

```
print("title", Collections.<TimeSeriesData>emptyList());
```

EXERCISES 8 – 9

Weitere Infos

- https://www.baeldung.com/java-generics
- http://tutorials.jenkov.com/java-generics/index.html
- https://www.javatpoint.com/generics-in-java
- https://www.torsten-horn.de/techdocs/java-generics.htm