Vorlesung

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Programmieren I und II

Unit 9

Generische Datentypen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Prof. Dr. rer. nat.
Nane Kratzke

Praktische Informatik und
betriebliche Informationssysteme

- **Raum: 17-0.10**
- **Tel.:** 0451 300 5549
- **Email: kratzke@fh-luebeck.de**

@NaneKratzke

Updates der Handouts auch über Twitter #prog_inf und #prog_itd

Units

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Unit 1 Einleitung und Grundbegriffe	**Unit 2** Grundelemente imperativer Programme	**Unit 3** Selbstdefinierbare Datentypen und Collections	**Unit 4** Einfache I/O Programmierung
Unit 5 Rekursive Programmierung und rekursive Datenstrukturen	**Unit 6** Einführung in die objektorientierte Programmierung und UML	**Unit 7** Konzepte objektorientierter Programmiersprachen	**Unit 8** Testen (objektorientierter) Programme
Unit 9 Generische Datentypen	**Unit 10** Objektorientierter Entwurf und objektorientierte Designprinzipien	**Unit 11** Graphical User Interfaces	**Unit 12** Multithread Programmierung

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
3

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Worum geht es in dieser Unit ???

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
4

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Die generischen ??? und das Geheimnis der spitzen Klammer

```
A Generic<? super Concept> Story
```


Abgedeckte Ziele dieser UNIT

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Kennen existierender Programmierparadigmen und Laufzeitmodelle	Sicheres Anwenden grundlegender programmiersprachlicher Konzepte (Datentypen, Variable, Operatoren, Ausdrücke, Kontrollstrukturen)	Fähigkeit zur problemorientierten Definition und Nutzung von Routinen und Referenztypen (insbesondere Liste, Stack, Mapping)	Verstehen des Unterschieds zwischen Werte- und Referenzsemantik
Kennen und Anwenden des Prinzips der rekursiven Programmierung und rekursiver Datenstrukturen	Kennen des Algorithmus-begriffs, Implementieren einfacher Algorithmen	Kennen objektorientierter Konzepte Datenkapselung, Polymorphie und Vererbung	Sicheres Anwenden programmiersprachlicher Konzepte der Objektorientierung (Klassen und Objekte, Schnittstellen und Generics, Streams, GUI und MVC)
Kennen von UML Klassendiagrammen, sicheres Übersetzen von UML Klassendiagrammen in Java (und von Java in UML)	Kennen der Grenzen des Testens von Software und erste Erfahrungen im Testen (objektorientierter) Software	Sammeln erster Erfahrungen in der Anwendung objektorientierter Entwurfsprinzipien	Sammeln von Erfahrungen mit weiteren Programmiermodellen und -paradigmen, insbesondere Mulithread Programmierung sowie funktionale Programmierung

Am Beispiel der Sprache JAVA

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

Zum Nachlesen …

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Kapitel 17

Generizität

17.1 Generische Klassen

17.2 Eigenständig generische Methoden

17.3 Bounded Typ-Parameter und Wildcards

17.4 Generische Schnittstellen

Themen dieser Unit

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Generizität	Bounded Types
• Typsicherheit • Type Erasure • Generizität in Vererbungsbeziehungen • Eigenständig generische Methoden	• Einschränkung der Generizität • Wildcards • Upper Bounds • Lower Bounds

Das Konzept der Generizität

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Generizität erlaubt es, die Definition von Klassen, Methoden und Schnittstellen mit Hilfe von „Typ-Platzhaltern“ (Typ-Parametern) durchzuführen.
- Dadurch werden parametrisierbare Elemente geschaffen, die im Programm mit konkreten Datentypen aufgerufen werden können.

Eine Klasse mit formalem Typ-Parameter ist eine **generische Klasse**. Der Typ-Parameter ist ein symbolischer Name, der eingeschlossen in **spitzen Klammern** nach dem Klassennamen angegeben werden kann.

```
class Punkt <T> {
  private T x;
  private T y;

  public Punkt (T x, T y) {
    this.x = x;
    this.y = y;
  }
}
```

```
Punkt<Integer> pi =
  new Punkt<Integer>(1,2);
```

```
Punkt<Double> pi =
  new Punkt<Double>(1.0,2.0);
```


Warum Generizität?

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Mit der Generizität von Klassen, Schnittstellen und Methoden werden die folgenden Ziele verfolgt:
  - **Höhere Typsicherheit:** Erkennen von Typ-Umwandlungsfehlern zur Kompilierzeit anstatt zur Laufzeit
  - **Wiederverwendbarkeit** von Code
  - **Vermeiden des expliziten Casts** beim Auslesen aus einer Collection aus Elementen vom Typ Object

Generische Klassen (I)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Ohne Generizität (vor JAVA 5)

Komplette Kopie (nur die Datentypen sind geändert)

```
class PunktInteger  {
  private  Integer x;
  private  Integer y;

  public  Punkt (Integer  x, Integer  y) {
    this.x = x;
    this.y = y;
  }
}
```

```
class PunktDouble  {
  private  Double x;
  private  Double y;

  public  Punkt (Double x, Double y) {
    this.x = x;
    this.y = y;
  }
}
```

Mit Generizität (seit JAVA 5)

Nur eine Klassendefinition notwendig. Datentypen sind parametrisiert.

```
class Punkt <T> {
  private T x;
  private T y;

  public Punkt (T x, T y) {
    this.x = x;
    this.y = y;
  }
}
```


Generische Klassen (II)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Generische Klassen können beliebig viele parametrisierte Datentypen haben:

```
class GenClass <T, R, S, ... , U, V, W> {
  T data1;
  R data2;
  ....
  W data_n;
}
```

- Die Benennung der Parameter kann dabei den üblichen JAVA-Konventionen für Bezeichner folgen.
- *D.h. Typparameter müssen nicht notwendig nur aus einem Buchstaben bestehen. Dies wird aus Gründen der Einfachheit in diesem Handout jedoch weiterhin so gemacht werden.*

Generische Klassen (III)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Durch die Nutzung **formaler Typ-Parameter** können Klassen unabhängig von einem speziellen Typ generisch definiert werden.
- Der formale Typ-Parameter wird bei der Verwendung der Klasse dann durch den gewünschten konkreten Datentyp ersetzt.
- Werden formale Typ-Parameter durch **aktuelle Typ-Parameter** (konkreter Referenzdatentyp) ersetzt, so handelt es sich um eine konkrete Ausprägung einer generischen Klasse, die auch **aktuell parametrisierte Klasse** genannt wird.
- Der aktuelle Typ-Parameter muss ein **Referenztyp** und darf kein primitiver Datentyp sein.
- Aktuell parametrisierte Klassen stehen auf einer Ebene in der Vererbungshierarchie. **Zwischen ihnen bestehen keine Vererbungsbeziehungen!**

Generische Klassen (IV)

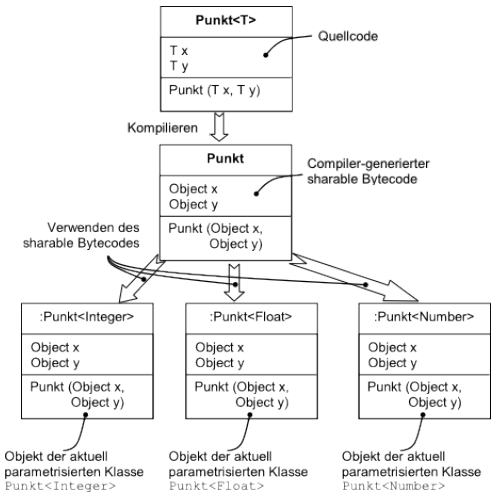
FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Shareable Bytecode

Der Compiler erzeugt bei der Übersetzung einer generischen Klasse nur eine einzige Bytecode-Datei, unabhängig davon, wie viele aktuell parametrisierte Klassen der generischen Klasse verwendet werden. Für die Klasse `Punkt` erzeugt der Compiler nur eine `Punkt.class` Datei.

Type Erasure

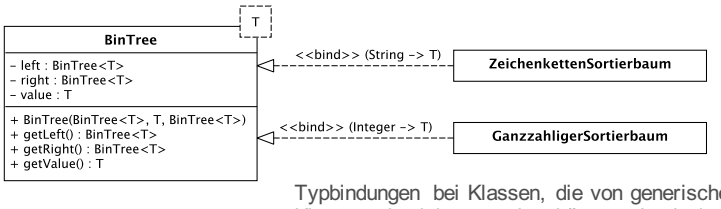
Beim Kompilieren werden alle Typ-Parameter `T` „ausradiert“ und durch die allgemeinste Klasse `Object` ersetzt.



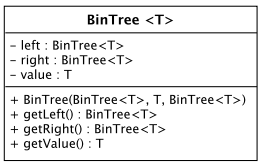
Generische Klassen (UML Notation)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Die UML Notation für generische Klassen sieht wie folgt aus. Der Typparameter wird als gestricheltes Rechteck in der oberen rechten Ecke dargestellt.



Typbindungen bei Klassen, die von generischen Klassen abgeleitet werden, können durch den Sterotyp <<bind>> ausgedrückt werden.



Häufig wird der Typparameter in generischen Klassen auch durch <T> gekennzeichnet. Dies ist aber streng genommen kein definiertes UML (wird aber von den meisten verstanden).

Type Erasure

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Durch Type Erasure werden beim Übersetzen einer generischen Klasse alle Vorkommen der formalen Typ-Parameter `T` ersetzt und im Rumpf der Klasse durch `Object` ersetzt.

```
class GenClass <T> {
  private T data;

  public T getData() {
    return data;
  }

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

```
class GenClass {
  private Object data;

  public Object getData() {
    return data;
  }

  public void setData(Object d) {
    data = d;
  }
}
```

- Der dann eigentlich erforderliche explizite Down-Cast müsste vom Programmierer selbst gemacht werden (wie von Collections bekannt). Dies erfolgt aber durch den Compiler vorgenommen transparenten Cast-Operator beim Aufruf von Methoden mit Rückgabetypen der parametrisierten Datentypen automatisch.

```
GenClass<Integer>  ref = new GenClass<Integer>();
Integer data = ref.getData();
// wird intern in die folgende Zeile übersetzt
Integer data = (Integer)ref.getData();
```

Generische Klasse und Vererbungsbeziehungen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Auch generische Klassen können Teil einer Vererbungshierarchie sein
- Folgende Fälle können auftreten:

Fall 1: Generische Klassen werden von anderen generischen Klassen abgeleitet

Fall 2: Generische Klassen werden von herkömmlichen Klassen abgeleitet

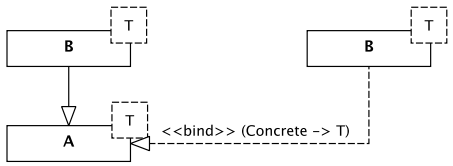
Fall 3: Herkömmliche Klassen werden von generischen Klassen abgeleitet

Fall 1:
Generische Klasse abgeleitet von generischer Klasse

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Erste Möglichkeit
Parameter wird durchgeschleift

Zweite Möglichkeit
Parameter wird ersetzt und
neuer Typparameter wird eingeführt




```
class B<T> extends A<T>  {

}
```

In diesem Fall ist der formale Typ-Parameter `T` der Klasse `B` der aktuelle Typ-Parameter der Klasse `A`. Wird nun Klasse `B` mit dem aktuellen Typ-Parameter `Concrete` instanziiert, so werden auch alle Vorkommen des formalen Typ-Parameters innerhalb der Klassendefinition von `A` ersetzt.

```
class B<T> extends A<Concrete>  {

}
```

Eine Definition dieser Form hat zur Folge, dass alle Vorkommen eines für die Klasse `A` definierten formalen Typ-Parameters beim Ableiten durch den Typ `Concrete` ersetzt werden.

Würde `B` nun mit einem weiteren Typ-Parameter `FurtherConcrete` instanziiert, so wird der formale Typ-Parameter nur noch in `B`, nicht aber mehr in `A` ersetzt (`A<Concrete>` wird sozusagen wie eine herkömmliche Klasse aufgefasst).

Fall 1.1 (Beispiel):
Generische Klasse abgeleitet von generischer Klasse

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Beispiel: Erste Möglichkeit

```
class B<T> extends A<T>  {
  T data2;
  T getData2() { return data2; }
}
```

```
class A<T> {
  T data1;
  T getData1() { return data1; }
}
```

ist im Falle folgender Instanziierung

```
B<Integer> b = new B<Integer>();
```

eigentlich zu lesen wie:

```
class B extends A {
  Integer data2;
  Integer getData2() {
    return data2;
  }
}
```

```
class A {
  Integer data1;
  Integer getData1() {
    return data1;
  }
}
```


Fall 1.2 (Beispiel):
Generische Klasse abgeleitet von generischer Klasse

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Beispiel: Zweite Möglichkeit

```
class B<T> extends A<Double>  {
  T data2;
  T getData2() { return data2; }
}
```

```
class A<T> {
  T data1;
  T getData1() { return data1; }
}
```

ist im Falle folgender Instanziierung

```
B<Integer> b = new B<Integer>();
```

eigentlich zu lesen wie:

```
class B extends A {
  Integer data2;
  Integer getData2() {
    return data2;
  }
}
```

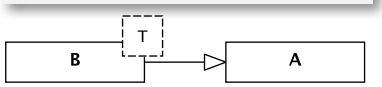
```
class A {
  Double data1;
  Double getData1() {
    return data1;
  }
}
```

Fall 2:
Generische Klasse erweitert herkömmliche Klasse

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
class B<T> extends A  {

}
```



Der Umstand, dass eine generische Klasse `B` von einer herkömmlichen Klasse `A` abgeleitet wird, hat keinen Einfluss auf den Code der herkömmlichen Klasse.

Die generische Klasse `B` erbt ganz herkömmlich nicht-generische Eigenschaften (Datenfelder und Methoden) der herkömmlichen Klasse `A`.

Beispiel:

```
class A {
  ConcreteType  data1;
  ConcreteType  getData1() {
    return data1;
  }
}
```

```
class B<T> extends A  {
  T data2;
  T getData2() {
    return data2;
  }
}
```

ist im Falle folgender Instantiierung

```
B<Integer>  b = new B<Integer>();
```

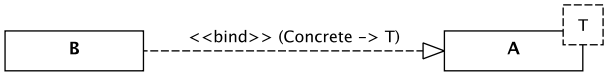
zu lesen wie folgt:

```
class B extends A  {
  Integer  data2;
  Integer  getData2() {
    return data2;
  }
}
```

An der Klasse `A` (da nicht parametrisiert) ändert sich nichts.

Fall 3:
Herkömmliche Klasse leitet von generischer Klasse ab

FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```
class B extends A<Concrete>  {

}
```

Dies kann nur geschehen, wenn beim Ableiten der formale Typ-Parameter der generischen Klasse `A` durch einen aktuellen Typ-Parameter gesetzt wird.

Andernfalls müsste `B` auch generisch sein

Beispiel:

```
class A<T> {
  T data1;
  T getData1() {
    return data1;
  }
}
```

```
class B extends A<Stack>
  Vector  data2;
  Vector  getData2() {
    return data2;
  }
}
```

ist zu lesen wie folgt:

```
class A {
  Stack  data1;
  Stack  getData2() {
    return data2;
  }
}
```

```
class B extends A {
  Vector  data2;
  Vector  getData2() {
    return data2;
  }
}
```

Eigenständig generische Methoden

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Klassenmethoden, Instanzmethoden und Konstruktoren können als eigenständige Methoden in einer Klasse existieren, **ohne dass die Klasse selbst generisch ist**.
- Die **Gültigkeit** der durch die Typ-Parameter-Sektion bekannt gemachten formalen Typ-Parameter bezieht sich nicht wie bei einer generischen Klasse auf die gesamte Klasse, sondern nur auf die entsprechende **Methode/Konstruktor**.

```
public <T> RetTyp genericMethod(T  p) {
  ...
}
```

- Üblicherweise setzt man generische Methoden bei Hilfsklassen ein, um ein und dieselbe Methode für verschiedene Typ-Parameter zu verwenden. Algorithmen sollen unabhängig vom Datentyp der Objekte sein, auf denen er ausgeführt wird.

Miniübung:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Entwickeln sind nun eine **eigenständig generische** Methode **`invert`**, die beliebige Listen **typsicher** umdreht und bspw. wie folgt aufgerufen werden kann (sie dürfen in der Methode `invert` keine Type Castings verwenden und die Methode `invert` nicht überladen):

```
List<String>  testString  = new LinkedList<String>();
List<Integer>  testInt = new LinkedList<Integer>();

for (int i = 1; i <= 100; i++) {
   testString.add("Test  " + i);
   testInt.add(i);
}

System.out.println(invert(testString));
System.out.println(invert(testInt));
```

Miniübung:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Denkbare Lösung:

Miniübung:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Entwickeln sind nun eine generische Klasse **Inverter**, die beliebige Listen **typsicher** umdreht und bspw. wie folgt aufgerufen werden kann (sie dürfen wieder keine Type Castings verwenden):

```
List<String>  testString  = new LinkedList<String>();
List<Integer>  testInt = new LinkedList<Integer>();

for (int i = 1; i <= 100; i++) {
   testString.add("Test  " + i);
   testInt.add(i);
}

Inverter<String>  i1 = new Inverter<String>(testStrings);
Inverter<Integer>  i2 = new Inverter<Integer>(testInts);

System.out.println(i1.invert());
System.out.println(i2.invert());
```

Miniübung:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Denkbare Lösung:

Themen dieser Unit

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Generizität

- Typsicherheit
- Type Erasure
- Generizität in Vererbungsbeziehungen
- Eigenständig generische Methoden

Bounded Types

- Einschränkung der Generizität
- Wildcards
- Upper Bounds
- Lower Bounds

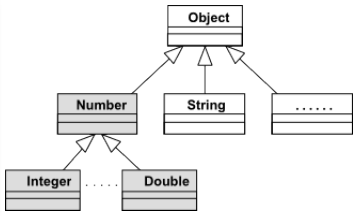
Bounded Typ-Parameter

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Ein Typ-Parameter innerhalb einer generischen Klasse kann eingeschränkt werden durch den Einsatz eines so genannten Bounded Typ-Parameters
- `T extends UpperBound`
- **Mit einem Bounded Typ-Parameter kann der zulässige Wertebereich von Typ-Parametern auf einen Teilbaum einer Klassenhierarchie eingeschränkt werden.**

```
class Punkt <T extends Number> {
  private T x;
  private T y;

  public Punkt (T x, T y) {
    this.x = x;
    this.y = y;
  }
}
```



Bei der Klasse Punkt würde einige Datentypen (z.B. `String`, `List`, `Stack`) als Typ-Parameter keinen Sinn ergeben, da die Datenfelder `x` und `y` immer numerische Werte sein sollten.

Dies lässt sich wie oben dargestellt ausdrücken. Ein unsinniges Objekt vom Typ `Punkt<Stack>` ist so nicht mehr möglich.

Type Erasure
bei Bounded Typ-Parametern

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Durch das Type Erasure werden beim Übersetzen einer generischen Klasse alle Vorkommen eines Bounded Typ-Parameter `T extends Type` ersetzt und im Rumpf der Klasse durch `Type` ersetzt.

```
class GenClass <T extends Type>
{
  private T data;

  public T getData() {
    return data;
  }

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

```
class GenClass
{
  private Type data;

  public Type getData() {
    return data;
  }

  public void setData(Type d) {
    data = d;
  }
}
```

- Ansonsten funktioniert das Verfahren wie beim unbounded Type Erasure. Der Compiler übernimmt transparent die eigentlich erforderlichen Down-Casts.

Unbounded Wildcard ?

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- **Um eine Referenzvariable definieren zu können, die auf Objekte beliebiger aktuell parametrisierter Klassen eines generischen Typs zeigen kann, existiert ein sogenannter Wildcard ?**
- Eine Referenzvariable **`Punkt<?> ref`** kann auf alle Objekte aktuell parametrisierter Klassen des generischen Typs **`Punkt<T>`** zeigen.
- Das `?` wird auch Unbounded Wildcard bezeichnet, weil es keine Einschränkungen gibt, durch welche konkreten Typ die Wildcard `?` ersetzt werden kann.
- Die **Wildcard `?`** steht im Gegensatz zu einem formalen Typ-Parameter `T` nicht stellvertretend für genau einen Typ, **sondern für alle möglichen Typen**.

```
Punkt<?> ref = new Punkt<Integer>(1, 2);
ref  = new Punkt<Double>(1.0, 2.0);
```

- **Beim Erzeugen von Arrays ist ein Wildcard zwingend zu nutzen.**

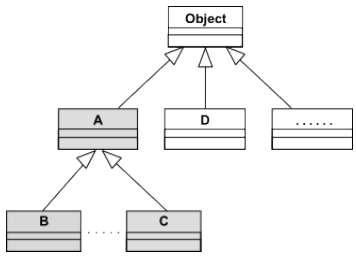
```
// funktioniert nicht
Punkt<Integer>[] fehler = new Punkt<Integer>[3];

// nur so richtig
Punkt<?>[] richtig = new Punkt<Integer>[3];
```

Upper Bound Wildcard

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Ähnlich wie sich Typ-Parameter einschränken lassen, lassen sich auch Wildcards einschränken.
- Die Upper Bound Wildcard
- **`<? extends UpperBound>`**
- legt eine **obere Schranke** für einen formalen Typ-Parameter fest.



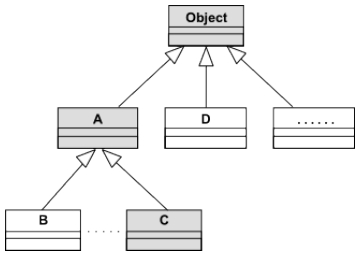

```
void methode(GenKlasse<? extends A> p) {
  ...
}
```

`methode()` akzeptiert damit Referenzen auf Objekte vom Typ `GenKlasse<T>`, wobei für den formalen Typ-Parameter `T` entweder der Typ `A` oder Subtypen von `A` erlaubt sind.

Lower Bound Wildcard

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Das Pendant zum Upper Bound Wildcard ist der Lower Bound Wildcard
- **<? super LowerBound>**
- Es legt eine **untere Schranke** für einen formalen Typ-Parameter fest.



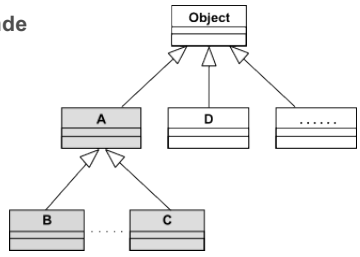

```
void methode(GenKlasse<? super C> p) {
  ...
}
```

`methode()` akzeptiert damit Referenzen auf Objekte vom Typ `GenKlasse<T>`, wobei für den formalen Typ-Parameter `T` entweder der Typ `C` oder Basistypen von `C` erlaubt sind.

Mini-Übung: Wildcard Raten

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Gegeben: Folgende Klassenhierarchie und folgende Methoden




```
void func(GenKlasse<? super B> p) {
  ...
}
```

```
void foo(GenKlasse<? extends A> p) {
  ...
}
```

Welche Aufrufe sind dann statthaft?

`func(new GenKlasse<A>());`	`// ok`	`foo(new GenKlasse<A>());`	`// ok`
`func(new GenKlasse<B>());`	`// ok`	`foo(new GenKlasse<B>());`	`// ok`
`func(new GenKlasse<C>());`	`// nok`	`foo(new GenKlasse<C>());`	`// ok`
`func(new GenKlasse<D>());`	`// nok`	`foo(new GenKlasse<D>());`	`// nok`

Generische Schnittstellen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Genauso wie Klassen können auch Schnittstellen generisch sein.
- Bspw. ist folgende Schnittstelle generisch:

```
public interface GenSchnittstelle <T> {
  public void methode1(T param);
  public void methode2();
}
```

- Die Implementierung einer generischen Schnittstelle durch eine Klasse kann nun auf zwei Arten erfolgen:
  - Die implementierende Schnittstelle ersetzt den formalen Typ-Parameter durch einen aktuell parametrisierten Typ-Parameter
  - Die implementierende Schnittstelle ersetzt den formalen Typ-Parameter nicht

Die zwei Möglichkeiten der Implementierung einer generischen Schnittstelle

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Erste Möglichkeit

```
class A implements IF<Concrete>
{

}
```

In diesem Fall müssen alle formalen Parameter der Schnittstelle `IF` durch aktuelle Typ-Parameter `Concrete` in der Klasse `A` ersetzt werden.

Die Klasse `A` ist eine herkömmliche Klasse.

Zweite Möglichkeit

```
class A<T> implements IF<T>
{

}
```

In diesem Fall implementiert, die Klasse A alle Methoden der generischen Schnittstelle IF und macht den formalen Typ-Parameter in ihrer Deklaration bekannt.

Die Klasse `A` wird so selbst generisch.

Mini-Übung:
Implementierung einer generischen Schnittstelle mit Ersetzung des formalen Typ-Parameters

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Gegeben sei folgende Schnittstellen-Definition:

```
interface IF<T> {
  T data1;
  T getData1();
}
```

Dann ist im Falle folgender Klassendeklaration

```
class A implements IF<List>  {
  ...

}
```

die Klasse A wie zu implementieren?

```
class A implements IF<List> {
  List data1;
  List getData1() {
    return data1;
  }
}
```

Die Klasse A kann dann wie instanziiert werden?

```
A ref = new A();
// A ist nicht generisch, der
// „innere“ Typ ist durch die
// Implementierung vorgegeben.
```


Beispiel:
Implementierung einer generischen Schnittstelle ohne Ersetzung des formalen Typ-Parameters

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Gegeben sei folgende Schnittstellen-Definition:

```
interface IF<T> {
  T data1;
  T getData1();
}
```

Dann ist im Falle folgender Klassendeklaration ...

```
class A<T> implements IF<T>  {
  ...

}
```

... die Klasse A wie zu implementieren?

```
class A<T> implements IF<T> {
  T data1;
  T getData1() {
    return data1;
  }
}
```

Die Klasse A kann dann wie instanziiert werden?

```
A<Stack> ref1 = new A<Stack>();
A<Integer> ref2 = new A<Integer>();
A<MyType> ref3 = new A<MyType>();
// Klasse A ist generisch!!!
```

Schnittstellen und Bounds

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- UpperBounds werden bei formalen Typ-Parametern dazu genutzt, um aktuelle Typ-Parametrisierungen auf Teilbäume einer Klassenhierarchie einzuschränken.
- Sie haben ferner gelernt, dass in JAVA Klassen nur von einer Klasse abgeleitet werden können, aber beliebig viele Schnittstellen implementieren können.
- Beide Regeln gelten auch für die Definition von UpperBounds von formalen Typ-Parametern.

Definition von Bounds
mit Klassen und Schnittstellen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Eine Bound kann aus einer Klasse `Klasse` und beliebig vielen Schnittstellen `I1` bis `In` bestehen.
- Die Verknüpfung der einzenen Bounds findet über den logischen UND-Operator `&` statt.
- Die Deklaration einer solchen Bound für den formalen Typ-Parameter `T` lautet dann:

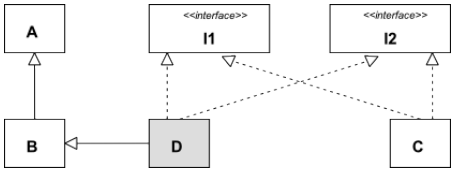
```
T extends Klasse & I1 & I2 & ... & In
```

- Diese Notation hat zur Konsequenz, dass der aktuelle Typ-Parameter, der den formalen Typ-Parameter `T` ersetzt,
  - von der Klasse `Klasse` abgeleitet ist
  - und ferner alle Schnittstellen `I1` bis `In` implementiert hat
- damit der Compiler die Ersetzung des formalen Typ-Parameters durch den aktuellen Typ-Parameter zulässt.

Mini-Übung:
Bounds mit Klassen und Schnittstellen-Raten

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Gegeben sei folgende Klassenhierarchie. Welche der Referenztypen `A`, `B`, `C` und `D` sind dann zulässig als aktuelle Typ-Parameter für die generischen Klassen `Generic1`, `Generic2` und `Generic3`?




```
class Generic1 <T extends A> {
  private T data;
  public T getData() { return data; }
  ...
}
```

```
A ist zulässig
B ist zulässig, da von A abgeleitet
C ist nicht zulässig, da nicht von A abgeleitet
D ist zulässig, da von A abgeleitet
```

```
class Generic2 <T extends A & I1 & I2> {
  private T data;
  public T getData() { return data; }
  ...
}
```

```
A nicht zulässig, implementiert nicht I1 und I2
B nicht zulässig, implementiert nicht I1 und I2
C nicht zulässig, nicht von A abgeleitet
D ist zulässig
```

```
class Generic3 <T extends I1 & I2> {
  private T data;
  public T getData() { return data; }
  ...
}
```

```
A nicht zulässig, implementiert nicht I1 und I2
B nicht zulässig, implementiert nicht I1 und I2
C ist zulässig
D ist zulässig
```


Mini-Übung: Type Erasure bei
UpperBounds mit Klassen und Schnittstellen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Es gilt die folgende Regel beim Type Erasure in JAVA: **Durch das Type Erasure werden beim Übersetzen einer generischen Klasse die formalen Typ-Parameter, die eine UpperBound besitzen, durch den ersten Typ der UpperBound ersetzt.**

- So werden bei `class G<T extends Klasse & I1>` alle Vorkommen von `T` durch `Klasse` ersetzt.
- Und bei `class G<T extends I1 & I2>` alle Vorkommen von `T` durch `I1` ersetzt.

Wie sehen nach dieser Regel dann die unten stehenden Type Erasures aus?

```
class Generic1 <T extends A> {
  private T data;
  public T getData() { return data; }
  ...
}
```

```
class Generic1 {
  private A data;
  public A getData() { return data; }
  ...
}
```

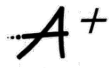
```
class Generic2 <T extends B & I1 & I2> {
  private T data;
  public T getData() { return data; }
  ...
}
```

```
class Generic2 {
  private B data;
  public B getData() { return data; }
  ...
}
```

```
class Generic3 <T extends I1 & I2> {
  private T data;
  public T getData() { return data; }
  ...
}
```

```
class Generic3 {
  private I1 data;
  public I1 getData() { return data; }
  ...
}
```

Zusammenfassung (Generizität)



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Konzept und Gründe für Generizität
- Shareable Byte Code und Type Erasure
- Generische Klassen in Vererbungsbeziehungen
- Generische Methoden
- Bounded Type Parameter und Type Erasure
- (Upper/Lower Bound) Wildcards
- Generische Schnittstellen
- UpperBounds mit (Klassen und) Schnittstellen und Type Erasure

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
43