



Christian Uhlig

Fortgeschrittene OOP SS19

- Fortgeschrittene OOP ist die Fortsetzung von PS2 (nicht von AuD)
- Fokus auf Konzepten der Programmierung (sprich: zur Formulierung von Software), relativ wenig auf Algorithmen
- Konzepte und ihre Anwendung am Beispiel von Java
- **Aber:** Überwiegend kein spezifischer Bezug zu objektorientierten Sprachen, d.h. die Konzepte sind vielfach auf andere Paradigmen übertragbar
- In den ersten zwei Wochen jeweils 2 Vorlesungseinheiten
 - Mi 10.04. 8:00 HS2 + Do 11.04. 12:30 HS2
 - Mi 17.04. 8:00 HS2 + Do 18.04. 12:30 HS2
- Übungsveranstaltung
 - Leitung durch Malte Heins und Helga Karafiat, Mailanfragen an foop@fh-wedel.de
 - Anmeldung ab **heute 19:30** bis **Mi 17.04. 8:00** im **Online-Sekretariat**
 - Einführungsveranstaltung mit Anwesenheitspflicht am **Mi 24.04. 15:30 (HS2)**

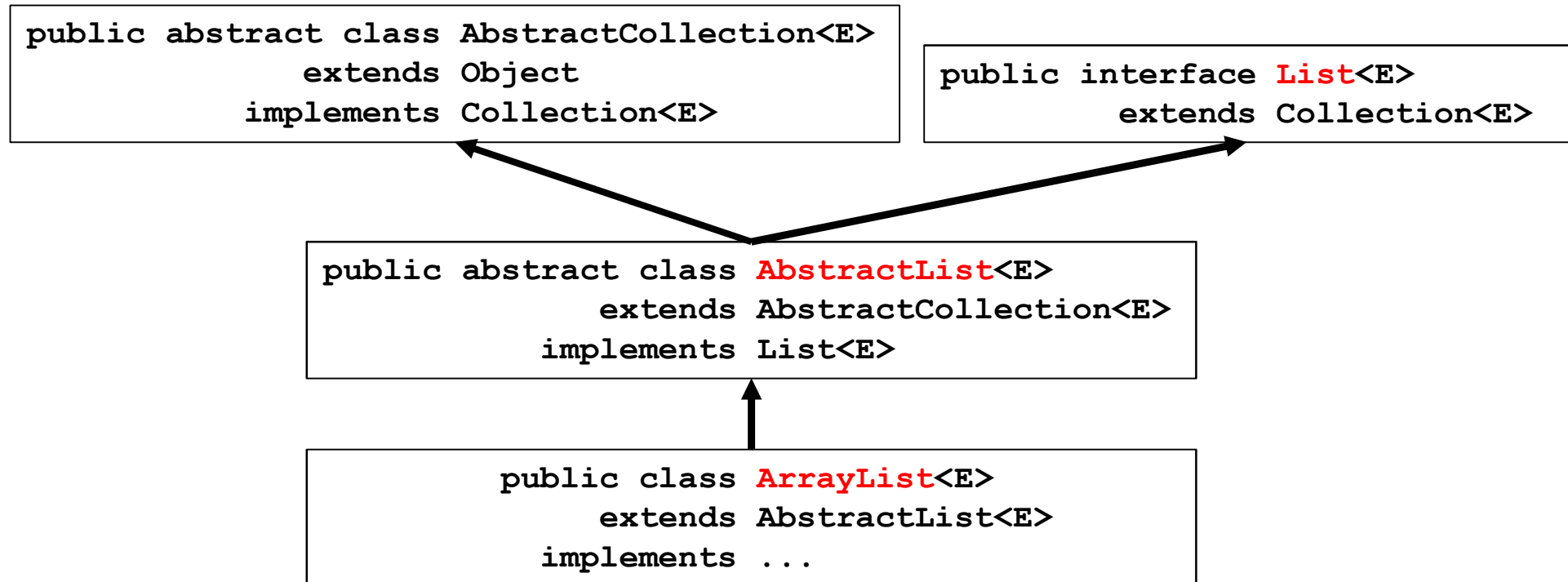
- Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, Alex
The Java Language Specification
Java SE 8 Edition
2015
- Lindholm, Tim; Yellin, Frank; Bracha, Gilad; Buckley, Alex
The Java Virtual Machine Specification
Java SE 8 Edition
2015
- Ullenboom, Christian
Java ist auch eine Insel
11. Auflage
Galileo Computing, 2014
- Meyer, Bertrand
Objektorientierte Softwareentwicklung
2. Auflage
Hanser Fachbuchverlag, 1990

- Bracha, Gilad
Generics in the Java Programming Language
2004
- Bloch, Joshua
Effective Java: A Programming Language Guide
2. Auflage
Addison-Wesley, 2008
- Louden, Kenneth C.
Programming Languages: Principles and Practice
2. Auflage
Thomson Learning, 2002

- 1. Fortgeschrittene Generics**
- 2. Verschachtelte Typen**
- 3. Funktionale Interfaces**
- 4. Funktionale Programmierung mit Java Streams**
- 5. Nebenläufigkeit**
- 6. Reflection & Annotations**

▪ Kurz vorweg...

- ...nachfolgend wird für Beispiele an vielen Stellen folgender Ausschnitt aus der **Java Collections API** genutzt:



- Allgemeines Verständnis von **Polymorphie** ("**Vielgestaltigkeit**")
 - In der Regel ist die **Anwendbarkeit eines Bezeichners** (typischerweise einer Operation) auf Werte **unterschiedlicher Typen** gemeint
 - Prägnantes Beispiel: `object.toString()` => auf jeden Referenztypen anwendbar, optional spezialisierte Implementierungen definierbar
 - `toString()` ist demzufolge "vielgestaltig" bzgl. ihrer Anwendbarkeit auf unterschiedlichste Referenztypen (**polymorphe Operation**)
 - Oder z.B. auch Vielgestaltigkeit einer Variablen, in OO-Sprachen kann eine Variable häufig zur Laufzeit auf Werte unterschiedlicher speziellerer Typen verweisen
 - Prägnantes Beispiel: `object o = 42;`
- Polymorphie ist immer Gegenstand von Konzepten des Typsystems
- Zweck ist insbesondere die **Wiederverwendbarkeit** bzw. Anwendbarkeit von Programmcode, im Falle von Java Generics auch **Typsicherheit**

■ Subtype-Polymorphie

- Existieren zum Typen eines Operationsparameters **Untertypen**, so ist die Operation bzgl. dieser Typhierarchie polymorph, d.h. jede für einen Typ `x` definierte Operation ist automatisch auch auf die Untertypen von `x` anwendbar
=> **1 Implementierung, n Typen**

- **Vererbung ermöglicht Subtype-Polymorphie**

Per Vererbung wird regelmäßig ein **Untertyp** der beerbten Klasse gebildet - so ist der erbende Typ `ArrayList` automatisch Untertyp von `AbstractList`. Die zum Obertypen (hier: `AbstractList`) definierten Methoden sind ohne Redefinition auch auf Untertypen (hier: `ArrayList`) anwendbar.

- **Notation für Untertyp-Beziehungen**

$X >_1 Y \Rightarrow X$ ist direkter Obertyp von Y (z.B. per `Y extends X`)

$X :> Y \Rightarrow X$ ist Obertyp von Y , wobei $X = Y$ gelten kann
(transitive & reflexive Hülle über $>_1$)

$X > Y \Rightarrow X$ ist echter Obertyp von Y , $X :> Y$ und $X \neq Y$

Eine Stufe
in der Relation
↳ direkter Ober-
/Untertyp

Indirekte Beziehung über mehrere Stufen

▪ Subtype-Polymorphie (Forts.)

- **Beispiel-Aussagen:** `AbstractList >1 ArrayList`, `List :> ArrayList`,
`List > ArrayList`, `ArrayList :> ArrayList`, `Object :> ArrayList`
Falsch: `Object >1 ArrayList`, `ArrayList :> AbstractList`, `List > List`
- **Untertypen-Beziehung führt zu Zuweisungskompatibilität:** Werte eines Typen `x` können anstelle von Werten eines beliebigen Obertypen `y :> x` verwendet werden

```
public static int sum(AbstractList<Integer> l) {  
    int result = 0;  
    int length = l.size();  
    for(int i = 0 ; i < length ; i++)  
        result += l.get(i);  
    return result;  
}  
  
AbstractList<Integer> al = new ArrayList<Integer>();  
/* ... */  
int s = sum(al);
```

Die statische Methode `sum()` kann also auf Werte sämtlicher Untertypen von `AbstractList` angewendet werden; `sum()` ist damit polymorph.

▪ Subtype-Polymorphie (Forts.)

- Auch die Deklaration von **Instanzmethoden** nutzt Subtype-Polymorphie aus:

```
public abstract class AbstractList<E>
    extends AbstractCollection<E>
    implements List<E> {

    /* ... */
    public int count(E elem) { /* impliziter Parameter this (Typ AbstractList) */
        int result = 0;
        int length = this.size();
        for(int i = 0 ; i < length ; i++)
            result += ( this.get(i) == elem ? 1 : 0 );
        return result;
    }
    /* ... */
}
```

- `count()` polymorph bzgl. Untertypen von `AbstractList`

- **Weiteres Anwendungsbeispiel:** Übertragung einer vorhandenen Container-Implementierung (`PairInt`) auf einen anderen Elementtypen (z.B. `String`)

```
public class PairInt {  
    private Integer left, right;  
  
    public PairInt(Integer left, Integer right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    public Integer l() {  
        return this.left;  
    }  
  
    public Integer r() {  
        return this.right;  
    }  
}
```

- **Beobachtung:** Prinzipiell unnötige Beschränkung des Elementtypen, da der Typ von `left` bzw. `right` für die Implementierung unerheblich ist

- Mögliche Flexibilisierung: Alle (Referenz-)Typen zulassen
=> Nutzung von **Subtype-Polymorphie**

```
public class PairObject {  
    private Object left, right;  
  
    public PairObject(Object left, Object right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    public Object l() {  
        return this.left;  
    }  
  
    public Object r() {  
        return this.right;  
    }  
}
```

- **Sehr flexibel:** In einem `PairObject` kann nun jeder Referenztyp (jeder Untertyp von `Object`) ohne Implementierungsanpassung gespeichert werden

```
PairObject p = new PairObject(23, 42);  
Integer i = p.l();  
/* Compiler-Fehler: Object -> Integer (narrowing reference conversion) */  
Integer i = (Integer)p.l();  
/* kompilierbar, aber Laufzeitfehler möglich (ClassCastException) */  
String s = (String)p.r();  
/* kompilierbar, aber Laufzeitfehler möglich (ClassCastException) */
```

- **Nachteil:** Zur Übersetzungszeit kann das Typsystem zu den Elementen eines `PairObject`-Objektes nur garantieren, dass sie vom Typ `object` sind
- **Explizite Casts** erforderlich
 - Da ein Rückgabewert vom Typ `object` zur Laufzeit jedem Referenztypen entsprechen kann, ist die Zuweisung auf etwas spezielleres (hier: `integer` bzw. `string`) **unsicher**
 - Der cast macht die Zuweisung nicht sicher, sondern bringt den Compiler zum Schweigen - erst zur Laufzeit stellt sich heraus, ob der Codeabschnitt für einen gegebenen Anwendungsfall ausführbar ist

```
PairObject p = new PairObject(23, 42);  
Integer i = p.l();  
/* Compiler-Fehler: Object -> Integer (narrowing reference conversion) */  
Integer i = (Integer)p.l();  
/* kompilierbar, aber Laufzeitfehler möglich (ClassCastException) */  
String s = (String)p.r();  
/* kompilierbar, aber Laufzeitfehler möglich (ClassCastException) */
```

- **Also:** Der Typ `object` macht die `PairObject`-Klasse sehr flexibel, das Wissen und die Verantwortung für die konsistente Nutzung hinsichtlich der Laufzeittypen der gespeicherten Elemente liegt aber beim Programmierer
- Sollen also z.B. wie bei `PairInt` `Integer`-Objekte verwaltet werden, so ist dies zwar möglich, aber die Typprüfung durch den Compiler geht verloren
- Also eine `Pair`-Implementierung für jeden gewünschten Elementtypen, z.B. noch `PairString`?
Sicher nicht: Keine Ausnutzung von Polymorphie, massive Codeverdopplung (nur der Elementtyp unterscheidet sich)

```
public class Pair<E> {  
    private E left, right;  
  
    public Pair(E left, E right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    public E l() {  
        return this.left;  
    }  
  
    public E r() {  
        return this.right;  
    }  
}
```

Deklaration eines Typparameters unter Einführung der Typvariable E

Verwendung der Typvariablen E im Rahmen der Klasse Pair<E>

- Der Typparameter **E** steht für einen beliebigen Referenztypen, der für jede Verwendung des Typen **Pair<E>** individuell festgelegt wird

- Für den deklarierten **Typparameter** wird bei Verwendung der Klasse ein Referenztyp als **Typargument** angegeben:

```
Pair<Integer> pi = new Pair<Integer>(23, 42);

Integer i = pi.l();
/* Signatur der Methode für diesen Aufruf "public Integer l()"
=> ohne cast sicher und kompilierbar */

String s = (String)pi.r();
/* Signatur der Methode für diesen Aufruf "public Integer r()"
=> nicht kompilierbar: Der Laufzeittyp eines Ausdrucks vom statischen Typ Integer
kann nie String sein */

Pair<String> ps = new Pair<String>("Dreiundzwanzig", "Zweiundvierzig");
String s = ps.l();
/* Signatur der Methode für diesen Aufruf "public String l()" => sicher */
```

- Die Vorkommen der Typvariablen (im Beispiel `E`) werden in den Typen der Instanzvariablen und Methoden des entsprechenden Ausdrucks (`pi`, `ps`) durch das Typargument (hier `Integer` und `String`) ersetzt, d.h. die Signatur von z.B. `pi.l()` entspricht dann `public Integer l()` statt `public E l()`

- Vergleich nicht-generische und generische Klasse:

```
PairObject po      = new PairObject(23, "Text");  
PairInt pi         = new PairInt(23, 42);  
  
Pair<Object> pgo    = new Pair<Object>(23, "Text");  
Pair<Integer> pgi   = new Pair<Integer>(23, 42);
```

```
Integer i = pi.l();  
Integer ig = pgi.l();
```

```
Object o = po.l();  
Object og = pgo.l();
```

```
Integer i = (Integer)po.l();  
Integer ig = (Integer)pgo.l();
```

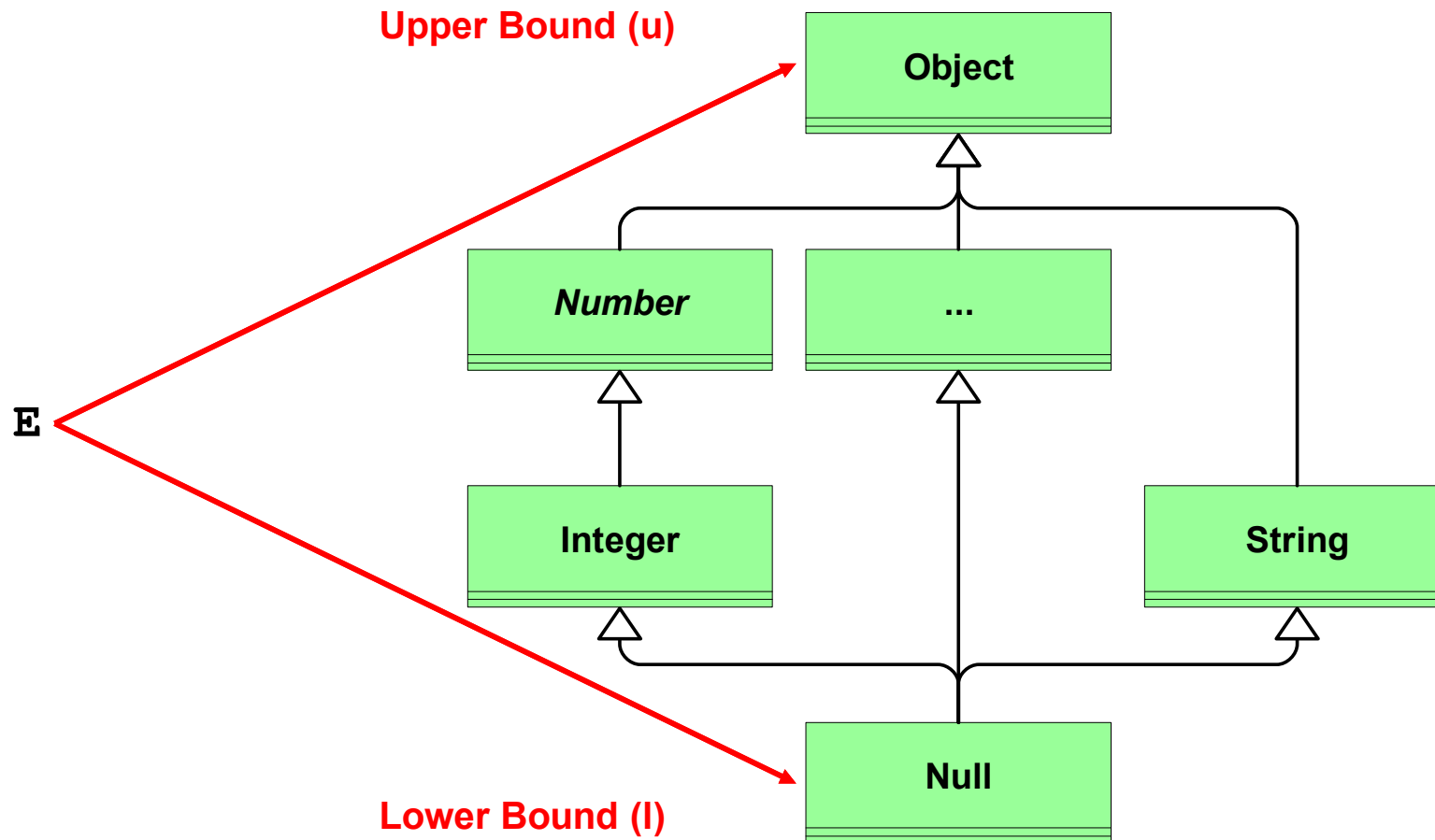
- Ein parametrisierter generischer `Pair`-Typ verhält sich genauso wie ein nicht-generischer `Pair`-Typ mit dem entsprechenden Elementtyp
- Es ist nur noch die Deklaration **eines** generischen Typen `Pair<E>` erforderlich. Soweit andere Typen für die `Pair`-Elemente benötigt werden, genügt eine entsprechende Parametrisierung des bestehenden generischen Typen.
- **Konzept: Parametrische Polymorphie (auch: **Generizität**)**
 - Der Typ `Pair<E>` und seine Operationen sind polymorph (auch: generisch), da sie ohne Neuimplementierung für unterschiedliche "tatsächliche" Typen parametrisiert werden können (**parametrisierte Typen**)

- Für Programmcode, der einen generischen Typen wie `Pair<E>` verwendet, ist der Typparameter `E` ein "Platzhalter", an dessen Stelle ein Typargument treten muss – dieses ersetzt jedes nach außen sichtbare Vorkommen des Typs `E`

	<code>Pair<Integer></code>
<code>public Pair(E left, E right)</code>	<code>=> public Pair(Integer left, Integer right)</code>
<code>public E l()</code>	<code>=> public Integer l()</code>
<code>public E r()</code>	<code>=> public Integer r()</code>

- Welche Typen dürfen für `E` als Typargument angegeben werden?
 - `E` repräsentiert eine **Menge** von zulässigen Typen, die durch eine obere Schranke `u` (**upper bound**) und eine untere Schranke `l` (**lower bound**) eingegrenzt wird
 - Obere Schranke `u`: **Allgemeinster** zulässiger Typ, standardmäßig `Object`
 - Untere Schranke `l`: **Speziellster** zulässiger Typ, grundsätzlich der **Null-Typ** (der nur die Null-Referenz enthält) *(Eingeführt um Untertypen für jeden anderen Typen zu haben)*
 - Für einen Typparameter `E` kann jeder Referenztyp `x` mit `u :> x ∧ x :> l` angegeben werden (standardmäßig also jeder Referenztyp)

- Welche Typen dürfen für E als Typargument angegeben werden? (Forts.)



▪ Innerhalb der deklarierenden Klasse ist E ein Referenztyp

- Innerhalb der generischen Klasse, in der der **Typparameter** E deklariert wird, stellt E zugleich eine **Typvariable** dar, die als **Referenztyp** verwendet werden kann

```
public class Pair<E> {  
    private E left, right;  
    /* ... */  
}
```

- Der Referenztyp E muss dabei die gemeinsamen Eigenschaften aller zulässigen Typen $T \in E$ vereinen, d.h. es dürfen nur Verwendungen zugelassen werden, die für **jedes konkrete Typargument** möglich wären
- Zuweisung eines Ausdrucks vom Typ v auf eine Variable des Typs E
 - v muss E entsprechen oder ein **Untertyp** von E sein, d.h. $v <: E$
 - Um ein Untertyp von E zu sein, muss v Untertyp jedes zulässigen Typargumentes sein, d.h. $\forall T \in E : v <: T$.
 - Dies ist zwangsläufig nur dann gewährleistet, wenn v der unteren Schranke l entspricht oder ein Untertyp davon ist, also $v <: l$.
 - Da die untere Schranke immer der Null-Typ ist, kann einer Variablen vom Typ E nur folgendes zugewiesen werden:
 - Ein Ausdruck vom Typ E **oder**
 - die Null-Referenz

"Für jedes
 T von E ..."

- Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)
 - Zuweisung eines Ausdrucks vom Typ **v** auf eine Variable des Typs **E** (Forts.)

```
public class Pair<E> {  
    private E left, right;  
  
    /* ... */  
  
    this.left = null;  
this.left = new Object();  
this.left = new Integer(42);  
  
    /* ... */  
  
    public Pair(E left, E right) {  
        this.left = left;  
        this.right = right;  
    }  
}
```

Zuweisung des Null-Wertes

Wäre z.B. nicht sicher,
wenn **E** = **String** wäre

Zuweisung eines Ausdrucks vom Typ **E**

- **Innerhalb der deklarierenden Klasse ist E ein Referenztyp (Forts.)**
 - **Umgekehrt:** Zuweisung eines Ausdrucks vom Typ E auf eine Variable des Typs v **bzw.** Kompatibilität eines Ausdrucks vom Typ E
 - v muss E entsprechen oder ein **Obertyp** von E sein, d.h. $v :> E$
 - Um ein Obertyp von E zu sein, muss v Obertyp jedes zulässigen Typargumentes sein, d.h. $\forall T \in E : v :> T$.
 - Dies ist zwangsläufig nur dann gewährleistet, wenn v der oberen Schranke u entspricht oder ein Obertyp davon ist, also $v :> u$.
 - Ist die obere Schranke vom Typ `Object`, kann ein Ausdruck vom Typ E nur folgendem zugewiesen werden:
 - Einer Variablen vom Typ E **oder**
 - einer Variablen von einem Obertypen von `Object` (d.h. ausschließlich `Object` selbst)
 - Frage der Zuweisungskompatibilität entscheidet auch über die **verfügbaren Member an einem Ausdruck vom Typ E** – an einem Ausdruck vom Typ E sind also standardmäßig die Member von `Object` verfügbar

verfügbare Methoden →

- Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)
 - **Umgekehrt**: Zuweisung eines Ausdrucks vom Typ **E** auf eine Variable des Typs **v** bzw. Kompatibilität eines Ausdrucks vom Typ **E** (Forts.)

```
public class Pair<E> {  
    private E left, right;
```

```
    public E l() {  
        return this.left;  
    }
```

```
    /* ... */
```

```
    E value    = this.left;  
    Object o   = this.left;  
    Integer i = this.left;  
    String s   = this.left.toString();
```

```
    /* ... */
```

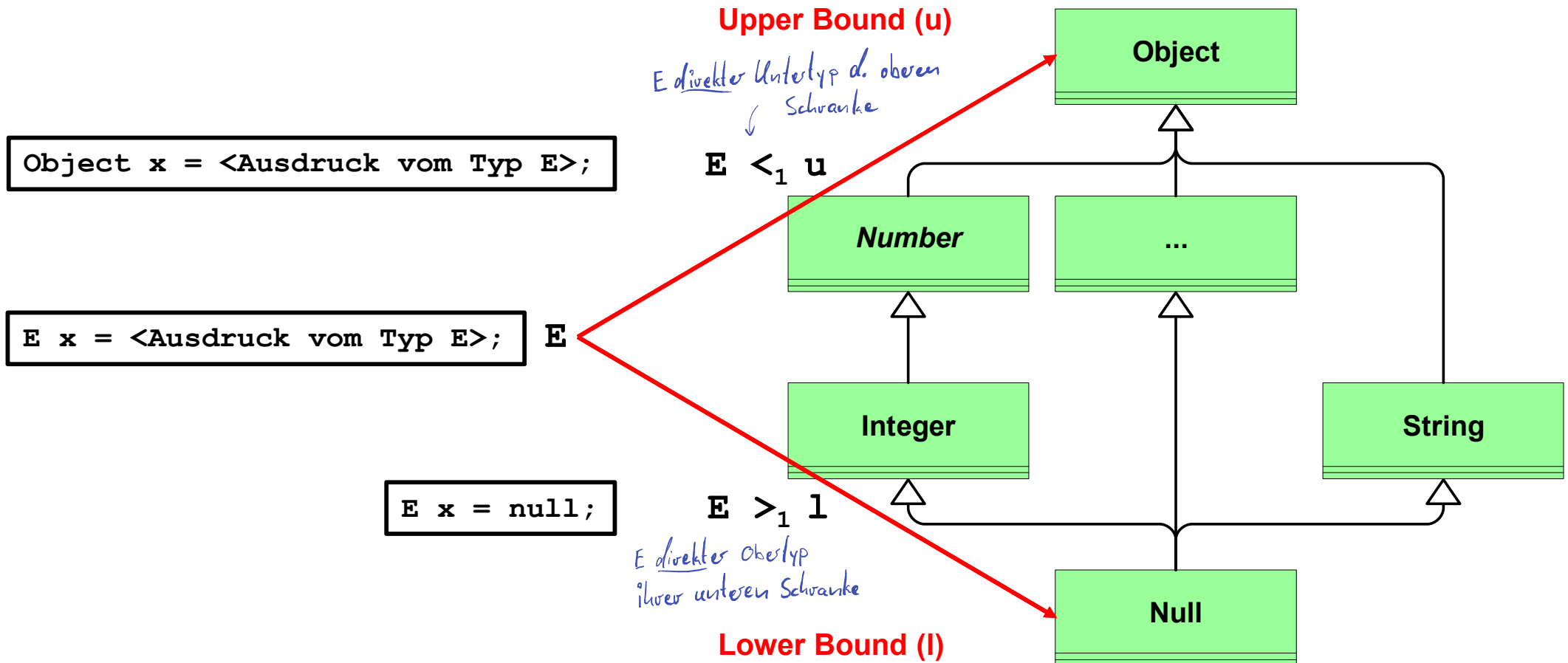
```
}
```

Typ **E** von **left** ist kompatibel
zum Rückgabetyp **E** von **l()**

Typ **E** von **left** ist kompatibel zu den Typen
E und **Object**, aber nicht zum Typ **Integer**

Alle Member von **Object** verfügbar
=> Aufruf von **toString()** möglich

- Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)



- Und wenn der linke und der rechte Wert eines Paares nicht zwingend vom selben Typ sein soll (was mit `PairObject` bereits möglich war)?
=> leicht durch Parametrisierung mit **zwei** Typparametern zu erreichen:

```
public class Pair<EL, ER> {  
    private EL left;  
    private ER right;  
  
    public Pair(EL left, ER right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    public EL l() {  
        return this.left;  
    }  
  
    public ER r() {  
        return this.right;  
    }  
}
```

Deklaration von zwei Typparametern unter Einführung der Typvariablen EL und ER

Verwendung der Typvariablen EL bzw. ER

- Bisherige Funktionalität (gleiche Typen für den linken und den rechten Wert) nach wie vor leicht zu gewährleisten:

```
Pair<Integer, Integer> p = new Pair<Integer, Integer>(23, 42);  
Integer i1 = p.l();  
Integer i2 = p.r();
```

- Zusätzlich besteht nun die Möglichkeit, unterschiedliche Typen anzugeben:

```
Pair<Integer, String> p = new Pair<Integer, String>(42, "Die Antwort");  
Integer i = p.l();  
/* Signatur der Methode für diesen Aufruf "public Integer l()" */  
String s = p.r();  
/* Signatur der Methode für diesen Aufruf "public String r()" */
```

- Die Parametrisierung von `Pair` war leicht möglich, da auf die enthaltenen Werte `left` / `right` nicht zugegriffen wurde (z.B. Methodenaufrufe)
- Wenn dem aber nun nicht so ist? Nicht-generisches Ausgangsbeispiel:

```
public class PairInt {  
    private Integer left, right;  
  
    /* ... */  
  
    public Integer min() {  
        return (this.left.compareTo(this.right) <= 0) ? this.left : this.right;  
    }  
}
```

- Die neue Methode `min()` soll das kleinere der beiden Elemente zurückgeben - das aber setzt die Anwendung eines Vergleichsoperators voraus

- Ein Vergleichsoperator existiert nicht für jeden Typen, d.h. ein Paar kann nur noch Elemente von Typen speichern, für die eine solche Operation definiert ist
- Mögliche Implementierung ausgehend von `PairObject`:

```
public class PairMin {  
    private Comparable left, right;  
  
    public Pair(Comparable left, Comparable right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    /* ... */  
  
    public Comparable min() {  
        return (this.left.compareTo(this.right) <= 0) ? this.left : this.right;  
    }  
}
```

- Elemente müssen nun vergleichbar sein bzw. für die entsprechenden Typen muss eine `compareTo()`-Methode definiert sein

- Wieder Ausnutzung von **Subtype-Polymorphie**: Jedes Objekt, dessen Klasse das `Comparable`-Interface implementiert (also dessen Typ Untertyp von `Comparable` ist), kann hier verwendet werden

```
PairMin p = new PairMin(new StringBuilder("ABC"), new StringBuilder("DEF"));  
PairMin p = new PairMin("ABC", "DEF");  
String s = (String)p.min();  
/* kompilierbar, aber Laufzeitfehler möglich (ClassCastException) */  
Integer i = (Integer)p.min();  
/* kompilierbar, aber Laufzeitfehler möglich (ClassCastException) */
```

`StringBuilder` implementiert `Comparable` nicht und wird hier folglich zurückgewiesen

- Aber gleiches Problem wie zuvor**: Zu den Elementen eines Paares wird nur garantiert, dass für sie das Interface `Comparable` implementiert ist; davon abgesehen können sie von jedem beliebigen Typen sein

- Lösung im Rahmen eines generischen Typen: Angabe einer **expliziten** oberen Schranke (**upper bound**) für die zulässigen Typen eines Typparameters

```
public class Pair<E extends Comparable<E>> {  
    private E left, right;  
  
    public Pair(E left, E right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    /* ... */  
  
    public E min() {  
        return (this.left.compareTo(this.right) <= 0) ? this.left : this.right;  
    }  
}
```

- Wirkung einer **upper bound** "**extends x**": Für den entsprechenden formalen Typparameter können nur noch Typargumente angegeben werden, die **Untertypen von x** sind (einschließlich **x** selbst). Es wird also der **allgemeinste** auf die Typvariable **E** zuweisbare Referenztyp festgelegt.

```
public class Pair<E extends Comparable<E>> {  
    /* ... */  
}
```

- Auf unser Beispiel übertragen können nun also für den Typparameter **E** nur noch Referenztypen angegeben werden, für die das Interface **Comparable** (d.h. die Methode **compareTo()**) implementiert ist

```
Pair<Number> p = new Pair<Number>(23, 42);  
/* Compiler-Fehler: Number implementiert Comparable<Number> nicht */  
Pair<String> p = new Pair<String>("42", "Die Antwort");  
/* OK: String implementiert Comparable<String> */  
String s      = p.min();
```

- Warum **extends Comparable<E>**?
Comparable ist selbst ein generisches Interface:

```
interface Comparable<T> {  
    int compareTo(T o);  
}
```

Der Typparameter **T** wird hierbei genutzt, um zu **compareTo()** einen Typ für das Vergleichsobjekt **o** festzulegen

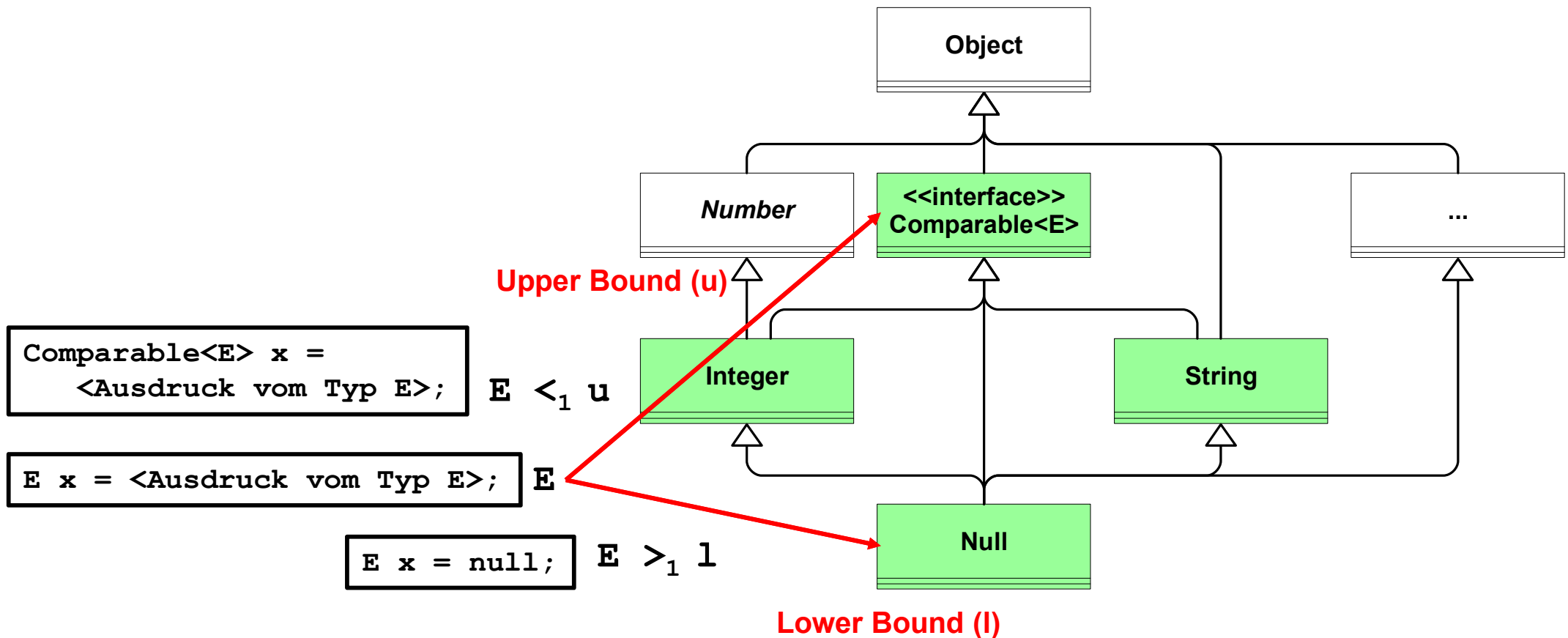
```
public class Pair<E extends Comparable<E>> {  
    /* ... */  
}
```

```
interface Comparable<T> {  
    int compareTo(T o);  
}
```

- Wir verwenden hier die Typvariable **E** von **Pair** als Typargument zur Parametrisierung ihrer eigenen **upper bound**. Intuitiv sind damit für **E** nur noch Referenztypen erlaubt, die mit **typgleichen** Werten verglichen werden können.
- Das ist naturgemäß die Regel, so implementiert z.B. die Klasse **Integer** das Interface **Comparable<Integer>** (im Übrigen ohne selbst generisch zu sein)
- Nur noch **ein** Typparameter: Für die **Pair**-Klasse mit **min()**-Methode müssen die beiden Elemente einen gemeinsamen Obertypen haben, welchen Rückgabetyt hätte sonst **min()**?

```
public class Pair<EL extends Comparable<ER>, ER extends Comparable<EL>> {  
    /* ... */  
    public ??? min();  
}
```


- Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)
 - Obere Schranke `Comparable<E>`, untere Schranke Null-Typ



▪ Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)

- Einer Variablen vom Typ **E** kann unverändert ein Ausdruck vom Typ **E** oder die Null-Referenz zugewiesen werden
- Ein Ausdruck des Typs **E** ist nun kompatibel zu jedem Obertypen von `Comparable<E>` (der oberen Schranke), kann also entsprechenden Variablen zugewiesen werden **und**
- für einen Ausdruck des Typs **E** stehen alle Member (Methoden, Variablen, ...) von `Comparable<E>` (der oberen Schranke) zur Verfügung (und damit auch von `Object`)

```
public class Pair<E extends Comparable<E>> {  
    private E left, right;  
    /* ... */  
  
    E left2          = this.left;  
    E right2         = null;  
    Object o         = this.left;  
    Comparable<E> c   = this.left;  
    Integer i         = this.left;  
    String s         = this.left.toString();  
    int cp           = this.left.compareTo(this.right);  
  
    /* ... */  
}
```

- **Mehrere Schranken:** Zu einem Typparameter können mehrere Referenztypen als Schranken definiert werden

```
public class Pair<E extends Number & Comparable<E>> {  
    /* ... */  
}
```

↳ Klassen immer zuerst genannt, es wird d. Schnittmenge gebildet.

Für die Schranken **2 bis n** sind dabei **nur Interfaces** zulässig

- **Bedeutung:** Für den formalen Typparameter können nur noch Typargumente angegeben werden, die Untertypen **aller** oberen Schranken sind

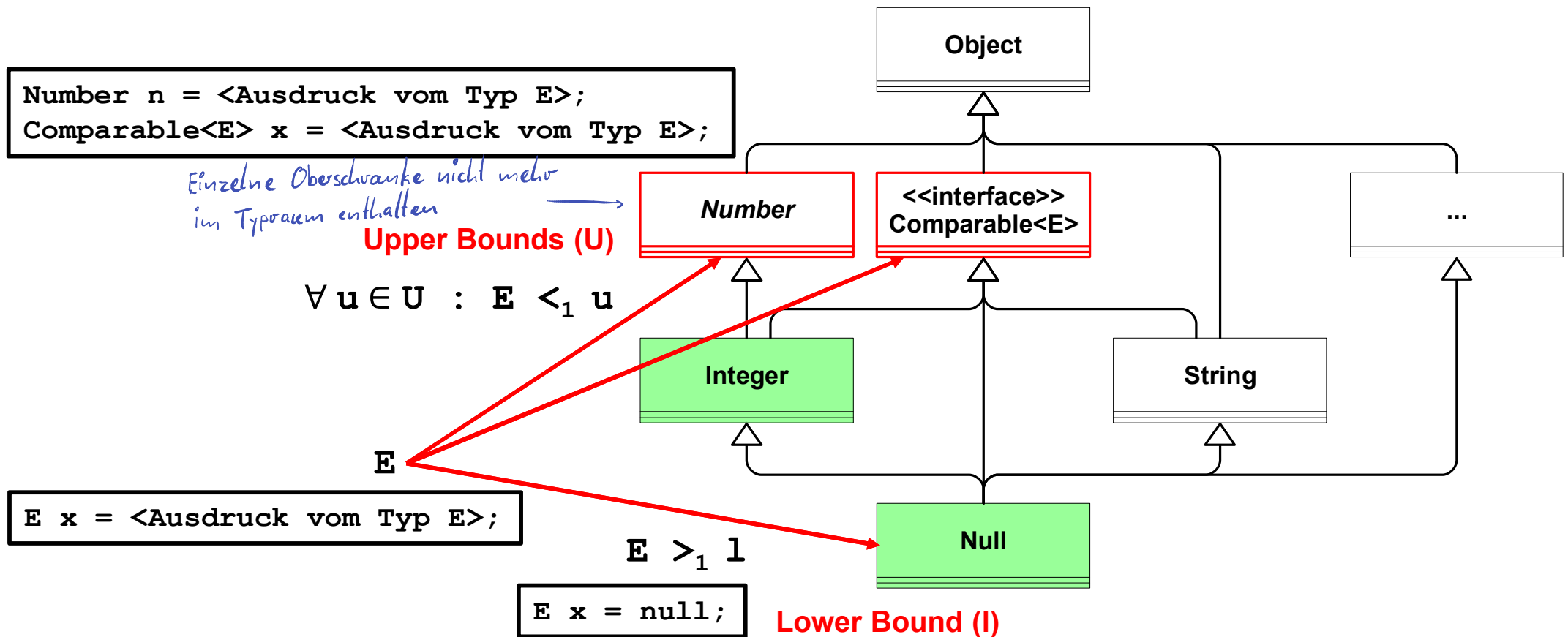
Formal: Sei U die Menge der oberen Schranken von E . Dann gilt:

$$\forall x \in E, u \in U : u :> x \wedge x :> l$$

- Bedeutung also hier: Es können nur `Pair`-Objekte von Typen gebildet werden, die Untertypen von `Number` sind und das `Comparable`-Interface implementieren ("vergleichbare Zahlen")

```
Pair<Integer> p = new Pair<Integer>(23, 42);  
/* OK: Integer erbt von Number und implementiert Comparable<Integer> */
```

- Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)
 - Obere Schranken `Number` & `Comparable<E>`, untere Schranke Null-Typ



▪ Innerhalb der deklarierenden Klasse ist **E** ein Referenztyp (Forts.)

- Einer Variablen vom Typ **E** kann unverändert ein Ausdruck vom Typ **E** **oder** die Null-Referenz zugewiesen werden
- Ein Ausdruck des Typs **E** ist nun kompatibel zu jedem Obertypen einer oberen Schranke (hier **Number** und **Comparable<E>**), kann also entsprechenden Variablen zugewiesen werden **und**
- für einen Ausdruck des Typs **E** stehen alle Member (Methoden, Variablen, ...) von **Comparable<E>** und **Number** (der oberen Schranken) zur Verfügung (und sämtlicher Obertypen)

```
public class Pair<E extends Number & Comparable<E>> {  
    private E left, right;  
    /* ... */  
  
    E left2          = this.left;  
    E right2         = null;  
    Object o         = this.left;  
    Number n         = this.left;  
    Comparable<E> c  = this.left;  
    Integer i        = this.left;  
    String s         = this.left.toString();  
    int cp           = this.left.compareTo(this.right);  
    int i            = this.left.intValue();  
  
    /* ... */  
}
```

■ Begrifflichkeiten

- **Generischer Typ:** Klasse oder Interface mit min. einem **formalen Typparameter**. Jeder generische Typ definiert eine Menge von **parametrisierten Typen**. Ein nicht-generischer Typ definiert genau einen Typen.

```
public class Example<E,F> {  
    E value;  
    E compute(F param) {  
        /* ... */  
    }  
}
```

- **Parametrisierter Typ:** Generischer Typ, für dessen Typparameter Referenztypen als **Typargumente** angegeben wurden

Für einen parametrisierten Typen tritt das Typargument an die Stelle von **E**.

```
Example<Integer, String> ex =  
    new Example<Integer, String>();  
  
/* ... */  
  
/* Integer compute(String param) */  
Integer i = ex.compute("42");
```

- **Typvariable:** Ein unqualifizierter Bezeichner, der mit einem Typparameter eingeführt wird und einen speziellen Referenztypen (im Sinne einer Menge von Typen) darstellt.

Die Variable bzw. der Typ **E** existiert nur innerhalb der deklarierenden Klasse bzw. des Interfaces (Sichtbarkeitsbereich).

parametrisierter Typ Bsp.: E extends Number ...

▪ Begrifflichkeiten (Forts.)

- **Typargument (tatsächlicher Typparameter):** Referenztyp, der im Rahmen eines parametrisierten Typen für eine Typvariable angegeben wird

```
Example<Integer, Comparable<String>> ex =  
    new Example<Integer, Comparable<String>>();
```

- Verwandtschaft zur Begriffswelt von Funktionen / Prozeduren / Methoden
 - Es werden formale Parameter deklariert, die jeweils eine Variable einführen, unter der ein für den Parameter übergebener Wert (Argument / tatsächlicher Parameter) zugreifbar ist.

```
public int example(int x, String y) {  
    return x + y.length();  
}
```

```
public class Example<E,F> {  
    /* ... */  
}
```

- **Wesentlicher Unterschied:** Statt für Werte stehen die Parameter für Typen
- Bei Aufruf einer Methode bzw. Parametrisierung des Typen sind Argumente anzugeben

```
int a = example(5, "some string");
```

```
Example<Integer, String> ex  
    = new Example<Integer, String>();
```

Deklaration generischer Typen (1)

*{ClassModifier} class Identifier **[TypeParameters]** [Superclass] [Superinterfaces] ClassBody*

*{InterfaceModifier} interface Identifier **[TypeParameters]** [ExtendsInterfaces] InterfaceBody*

- **TypeParameters**: Im Rahmen einer Klassen- oder Interface-Deklaration kann eine nicht-leere Liste von Typparametern angegeben werden
=> generische Klasse bzw. generisches Interface

TypeParameters = *< Identifier [TypeBound] {, Identifier [TypeBound] } >*

- **Identifier**: Die Typvariable zu einem Typparameter hat einen unqualifizierten Namen (wie eine lokale Variable)
- **TypeBound**: Für jede Typvariable können per Schlüsselwort **extends** obere Schranken (upper bounds) angegeben werden. **Zwei Alternativen** bestehen:
 - **extends ClassOrInterfaceType {& InterfaceType}**: mehrere per & verbundene **Klassen- oder Interfacetypen**, wobei nur die erste Schranke eine Klasse sein darf

```
public class Example<E extends Class & Interface1 & ...> { /* ... */ }
```

- **extends TypeVariable**: genau eine andere Typvariable

```
public class Example<E, F extends E> { /* ... */ }
```


- Implizit obere Schranke `object`, wenn keine Schranken angegeben werden
- Ein angegebenes Typargument muss Untertyp **aller** oberen Schranken sein.
- Der **Sichtbarkeitsbereich** einer Typvariable umfasst syntaktisch...
 - Typparameter der Klasse:
`public class Ex1<E extends Ex2<E>, F extends E>`
 - Typparameter von Oberklasse/Schnittstellen der Klasse:
`public class Ex1<E> extends Pair<E,E> implements Comparable<E>`
 - Klassenrumpf:
`public class Ex1<E> { E value; /* ... */ }`
- Typvariablen fast überall als Referenztyp verwendbar
 - **Deklaration von Typvariablen:** Schranken (`<E, F extends E>`) oder Parametrisierung von Schranken (`<E, F extends Comparable<E>>`)
 - **Klassendeklaration:** Parametrisierung von beerbten generischen Klassen und implementierten generischen Interfaces (`Ex1<E> extends Pair<E,E>`)
 - **Instanzvariablen:** Typ (`E value;`) oder Typparametrisierung (`List<E> value;`)
 - **Instanzmethoden:** Typ oder Typparametrisierung von Rückgabewert, Parametern und lokalen Variablen
(`public List<E> foo(E val, List<E> l) { E var; /* ... */ }`)

- Typvariablen fast überall als Referenztyp verwendbar (Forts.)
 - **nicht möglich:** Angabe als Oberklasse / Oberinterface (~~Ex1<E> extends E~~) bzw. zu implementierendes Interface (~~Ex1<E> implements E~~)
Kompiler muss schließlich zur Laufzeit d. Bytecode d. erbenenden erzeugen.
 - **nicht möglich:** Verwendung in statischem Kontext (statische Methoden, statische Variablen, statische Initializer-Blöcke)
- Eine Typvariable kann als Typargument verwendet werden, wenn sie eine Untermenge der zulässigen Typen des Typparameters darstellt

```
public class One<A extends Comparable<A>> {  
    /* ... */  
}
```

```
public class Two<E extends Number & Comparable<E>, F> {  
    /* ... */  
    One<F> obj = new One<F>();  
    One<E> obj = new One<E>();  
}
```

Nicht OK: Typparameter **A** lässt weniger Typen zu als **F** (unbeschränkt) (*comparable*)

OK: Typparameter **A** lässt alle Typen zu, die auch **E** zulässt

■ Korrekte Verwendungen

```
public class Example<E extends Number & Comparable<E>,
                    F extends E>
    extends Pair<E,F>
    implements Comparable<Example<E,F>> {
    private E        some_var;
    private List<E> further_var;

    public Example(E left, F right) {
        super(left, right);
    }

    public E getVal() {
        return this.some_var;
    }

    public List<E> getList() {
        List<E> result = new ArrayList<E>();
        result.add(this.some_var);
        return result;
    }

    public int compareTo(Example<E,F> other) {
        E val = other.some_var;
        return this.some_var.compareTo(val);
    }
}
```

▪ Fehlerhafte Verwendungen

```
public class Example<E,F>
    extends E           /* nicht erlaubt: Beerben einer Typvariablen */
    implements F {      /* nicht erlaubt: Implementieren einer Typvariablen */
        private static E static_var; /* nicht erlaubt: statischer Kontext */

        static {
            E useful = null;           /* nicht erlaubt: statischer Kontext */
        }

        /* nicht erlaubt: statischer Kontext */
        public static E compute(E val) {
            /* ... */
        }
    }
}
```

- Generische Typen (Klassen und Interfaces) sind per se nicht als Referenztypen verwendbar, sie müssen entweder **parametrisiert** oder als **Raw Type** genutzt werden:
- **Parametrisierter Typ**: Angabe einer **Parametrisierung** des generischen Typen, d.h. Angabe von Referenztypen zu allen formalen Typparametern

```
List<Integer> l = new ArrayList<Integer>();
```

- **Raw Type**: Angabe des generischen Typen ohne Typargumente

```
List l = new ArrayList();
```

Dient der Kompatibilität zu altem nicht-generischen Code und sollte vermieden werden

=> Die Angabe eines Raw Types führt zu einer Compiler-Warnung

- **Allgemeiner Grundsatz für Referenztypen:** Die Zuweisung eines Ausdrucks vom Typ R auf eine Variable vom Typ L ist zulässig, wenn R mit L **identisch** ist (**identity conversion**) oder R einen **echten Untertypen** von L darstellt (**widening reference conversion**). D.h. es muss gelten $R <: L$.
- Die **direkten Obertypen** von parametrisierten Typen werden durch die Oberklasse/-interfaces der generischen Klasse bzw. des generischen Interfaces bestimmt (z.B. gilt `AbstractList<E> >1 ArrayList<E>`)
- Bei Zuweisung eines **Raw Type** auf einen **parametrisierten** Typen wird eine **unchecked conversion** vorgenommen (Compiler-Warnung!)

```
ArrayList<Integer> l = new ArrayList<Integer>(); /* identity conversion */
AbstractList<Integer> l = new ArrayList<Integer>(); /* widening reference conver. */
List<Integer> l = new ArrayList<Integer>(); /* widening reference conver. */

ArrayList<Integer> l = new ArrayList(); /* unchecked conversion */
AbstractList<Integer> l = new ArrayList(); /* widening reference conversion
                                           + unchecked conversion */
List<Integer> l = new ArrayList(); /* widening reference Conversion
                                   + unchecked conversion */
```

- Und wie sieht es hiermit aus?

```
Object o          = new Integer(42) ;  
ArrayList<Object> l = new ArrayList<Integer>() ; /* OK ? */
```

- **Sieht intuitiv sinnvoll aus:**
`Integer` ist natürlich Untertyp von `Object` (jeder Referenztyp ist ein Untertyp von `Object`), also ist eine Liste von Ganzzahlen anschaulich auch eine Liste von beliebigen Objekten
- Gilt also $x <: Y \Rightarrow C<X> <: C<Y>$ (wobei `C` ein generischer Typ ist) ?
- Wenn dies so wäre, würde man die generischen Typen von Java als **kovariant** bezeichnen, d.h. die Typkompatibilität (in Abhängigkeit von einem Typargument) folgt einer Spezialisierung des Typargumentes **in gleicher Richtung** ("ko").

- **Betrachtung für** `Object` \rightarrow `Integer`

```
C<Object> c = new C<Integer>();
```

```
public class C<E> {  
    public E var;  
    public E foo(E param) { /* ... */ }  
}
```

- **1. Fall: In Ausdrücken**

```
Object val1 = c.foo(...);          /* Aufruf von:    Object foo(Object param) */  
Object val2 = c.var;               /* Zuweisung von: Object var          */
```

- Von `c` referenzierte Instanz hat Typ `C<Integer>`: Kompatibilität gewährleistet, wenn die Zuweisungen für Variablen und Methoden des Typen `C<Integer>` sicher bleiben

```
C<Object> c = new C<Integer>(); /* Kovarianz */  
Object val1 = c.foo(...);      /* Aufruf von:    Integer foo(Integer param) */  
Object val2 = c.var;           /* Zuweisung von: Integer var          */
```

- **Sicher**, da `Object` \rightarrow `Integer` gilt

- **Beispiel mit Ganzzahl-Liste:**

```
ArrayList<Object> lo = new ArrayList<Integer>(); /* Kovarianz */  
Object o = lo.get(0);                          /* OK, Object  $\rightarrow$  Integer */
```

- Aus einer Ganzzahl-Liste können Ganzzahlen (Typ `Integer`) entnommen und auf `Object` zugewiesen werden

- **Betrachtung für** `Object` `:` `Integer`

```
C<Object> c = new C<Integer>();
```

```
public class C<E> {  
    public E var;  
    public E foo(E param) { /* ... */ }  
}
```

- **2. Fall: Zuweisung auf Variablen**

```
c.foo( new Object() );           /* Aufruf von:    Object foo(Object param) */  
c.var = new Object();           /* Zuweisung auf: Object var      */
```

- Von `c` referenzierte Instanz hat Typ `C<Integer>`: Kompatibilität gewährleistet, wenn die Zuweisungen für Variablen und Methoden des Typen `C<Integer>` sicher bleiben

```
C<Object> c = new C<Integer>(); /* Kovarianz */  
c.foo( new Object() );         /* Aufruf von:    Integer foo(Integer param) */  
c.var = new Object();          /* Zuweisung auf: Integer var      */
```

- **Unsicher**, es müsste `Object` `<:` `Integer` gelten

- **Beispiel mit Ganzzahl-Liste:**

```
ArrayList<Object> lo = new ArrayList<Integer>(); /* Kovarianz */  
lo.add(new Integer(42));                        /* OK */  
lo.add(new Double(4.2));                        /* Fehler */  
lo.add(new Object());                          /* Fehler */
```

- In eine `Integer`-Liste können keine beliebigen `Object`-Untertypen eingefügt werden

- **Also:** Generische Typen in Java sind nicht kovariant. Anderenfalls wären Typfehler zur Laufzeit möglich.
- Und wie sieht es hiermit aus?

```
Object o          = new Integer(42);  
ArrayList<Integer> l = new ArrayList<Object>(); /* OK ? */
```

- **Sieht intuitiv nicht sinnvoll aus:** `Object` ist kein Untertyp von `Integer` (nicht jedes Objekt ist ein `Integer`). Es erscheint damit fehlerhaft, dass eine beliebige Liste von Objekten immer auch eine Liste von Ganzzahlen sein soll.
- **Trotzdem die Frage:** Gilt $x <: Y \Rightarrow c<x> :> c<Y>$ (wobei `c` ein generischer Typ ist) ?
- Wenn dies so wäre, würde man die generischen Typen von Java als **kontravariant** bezeichnen, d.h. die Typkompatibilität (in Abhängigkeit von einem Typargument) folgt einer Spezialisierung des Typargumentes **in entgegengesetzter Richtung** ("kontra").

- **Betrachtung für** `Object` \rightarrow `Integer`

```
C<Integer> c = new C<Object>();
```

```
public class C<E> {  
    public E var;  
    public E foo(E param) { /* ... */ }  
}
```

- **1. Fall: In Ausdrücken**

```
Integer val1 = c.foo(...);      /* Aufruf von:    Integer foo(Integer param) */  
Integer val2 = c.var;           /* Zuweisung von: Integer var */
```

- Von `c` referenzierte Instanz hat Typ `c<object>`: Kompatibilität gewährleistet, wenn die Zuweisungen für Variablen und Methoden des Typen `c<object>` sicher bleiben

```
C<Integer> c = new C<Object>(); /* Kontravarianz */  
Integer val1 = c.foo(...);     /* Aufruf von:    Object foo(Object param) */  
Integer val2 = c.var;          /* Zuweisung von: Object var */
```

- **Unsicher**, es müsste `Integer` \rightarrow `Object` gelten

- **Beispiel mit Ganzzahl-Liste:**

```
ArrayList<Integer> li = new ArrayList<Object>(42, "Text"); /* Kontravarianz */  
Integer i1 = li.get(0);                                     /* OK */  
Integer i2 = li.get(1);                                     /* Fehler */
```

- Nicht jedes Element einer `object`-Liste muss von einem `integer`-Untertypen sein

- **Betrachtung für** `Object` \rightarrow `Integer`

```
C<Integer> c = new C<Object>();
```

- **2. Fall: Zuweisung auf Variablen**

```
public class C<E> {  
    public E var;  
    public E foo(E param) { /* ... */ }  
}
```

```
c.foo( new Integer(42) );      /* Aufruf von:    Integer foo(Integer param) */  
c.var = new Integer(42);      /* Zuweisung auf: Integer var */
```

- Von `c` referenzierte Instanz hat Typ `c<object>`: Kompatibilität gewährleistet, wenn die Zuweisungen für Variablen und Methoden des Typen `c<object>` sicher bleiben

```
C<Integer> c = new C<Object>(); /* Kontravarianz */  
c.foo( new Integer(42) );      /* Aufruf von:    Object foo(Object param) */  
c.var = new Integer(42);      /* Zuweisung auf: Object var */
```

- **Sicher**, da `Integer` \leq `Object` gilt

- **Beispiel mit Ganzzahl-Liste:**

```
ArrayList<Integer> li = new ArrayList<Object>(); /* Kontravarianz */  
li.add(new Integer(42));                        /* OK */
```

- In eine `object`-Liste können beliebige Instanzen eingefügt werden, natürlich auch von `Integer`-Untertypen

- Die Probleme von Ko- und Kontravarianz sind im Grunde **spiegelverkehrt**
 - Mit Kovarianz kann das **Übergabe**n von Werten an Variablen oder Methodenparameter eines parametrisierten Typen scheitern, das **Entnehmen** aus Variablen oder Rückgabewerten hingegen wäre sicher
 - Mit Kontravarianz kann das **Entnehmen** von Werten aus Variablen oder Rückgabewerten eines parametrisierten Typen scheitern, das **Übergabe**n an Variablen oder Methodenparameter hingegen ist sicher
- **Daher:** Die Untertypen-Beziehungen parametrisierter Typen in Java sind hinsichtlich der Typargumente **invariant**
- Ein parametrisierter Typ **A<E>** ist nur dann Untertyp eines anderen parametrisierten Typen **B<F>**, wenn **A** Untertyp von **B** ist und die Typargumente **E** und **F** **identisch** sind:

```
List<Number> l = new ArrayList<Integer>(); /* Compilerfehler, Kovarianz! */  
List<Integer> l = new ArrayList<Number>(); /* Compilerfehler, Kontravarianz! */  
List<Integer> l = new ArrayList<Integer>(); /* OK, Invarianz! */
```

- Die **Invarianz** der generischen Typen ist allerdings in manchen Fällen hinderlich, wenn der Programmcode offensichtlich keine Probleme aufwirft:

```
public static void printFirst(List<Object> l) {  
    System.out.println("Erstes Element: " + l.get(0).toString());  
}  
  
/* ... */  
  
printFirst( new ArrayList<Object>() ); /* OK! */  
printFirst( new ArrayList<String>() ); /* Fehler: Kovarianz */
```

- **Nicht übersetzbar, obwohl offensichtlich sicher**
 - Egal welchen parametrisierten Typen die an `printFirst()` übergebene Liste hätte, die Elemente sind in jedem Fall zuweisungskompatibel zu `object` und unterstützen damit die `toString()`-Methode
 - **Kovarianz hier unkritisch**, da aus der Liste `l` nur Werte **entnommen**, aber keine an die Liste **übergeben** werden

- **Lösung:** Als Typargumente eines parametrisierten Typen können statt Referenztypen sogenannte **Wildcards** angegeben werden

```
public static void printFirst(List<?> l) {  
    System.out.println("Erstes Element: " + l.get(0));  
}  
  
/* ... */  
printFirst( new ArrayList<Object>() ); /* OK! */  
printFirst( new ArrayList<String>() ); /* OK! */
```

↳ List<?>: „Es gibt eine Schranke, ich kenne sie nur nicht.“
↳ List<Object>: „Man kann alles in eine Liste reinschreiben.“

- Eine Wildcard (geschrieben als `?`) stellt einen **unbekannten** Referenztypen dar
- **Idee:** Wie eine Typvariable steht eine Wildcard nicht für einen einzelnen Typen (z.B. `String`), sondern für eine **Menge von Typen**.
Das Typsystem soll dann nur solche Verwendungen (für Ausdrücke / Variablen) zulassen, die **mit jedem Typen der Menge** sicher wären.
- Ein mit einer Wildcard parametrisierter Typ ist **Obertyp für eine Menge von parametrisierten Typen**. Im Beispiel ist `ArrayList<String>` Untertyp von `List<?>` und kann dem Methodenparameter `l` zugewiesen werden.

- **Welchen Typen entspricht eine Wildcard als Typargument?**
 - Analog zu Typvariablen wird die Menge der "enthaltenen" Typen durch eine obere Schranke (**upper bound**) und eine untere Schranke (**lower bound**) definiert
 - **Obere Schranke u :** Wie bei unbeschränkten Typvariablen standardmäßig `object`
 - **Untere Schranke l :** Standardmäßig wie bei Typvariablen der Null-Typ
 - Eine Wildcard umfasst alle Referenztypen x mit $u :> x \wedge x :> l$
 - Wie eine unbeschränkte Typvariable umfasst eine Wildcard also im einfachsten Fall **sämtliche Referenztypen**

▪ Welchen Typen entspricht eine Wildcard als Typparameter? (Forts.)

- Im allgemeinen Fall umfasst eine Wildcard alle Referenztypen

- **Beispiel:**
Interface `List<E>`

Variablendeklaration

```
List<?> l;
```

Obere Schranke für ?
Object

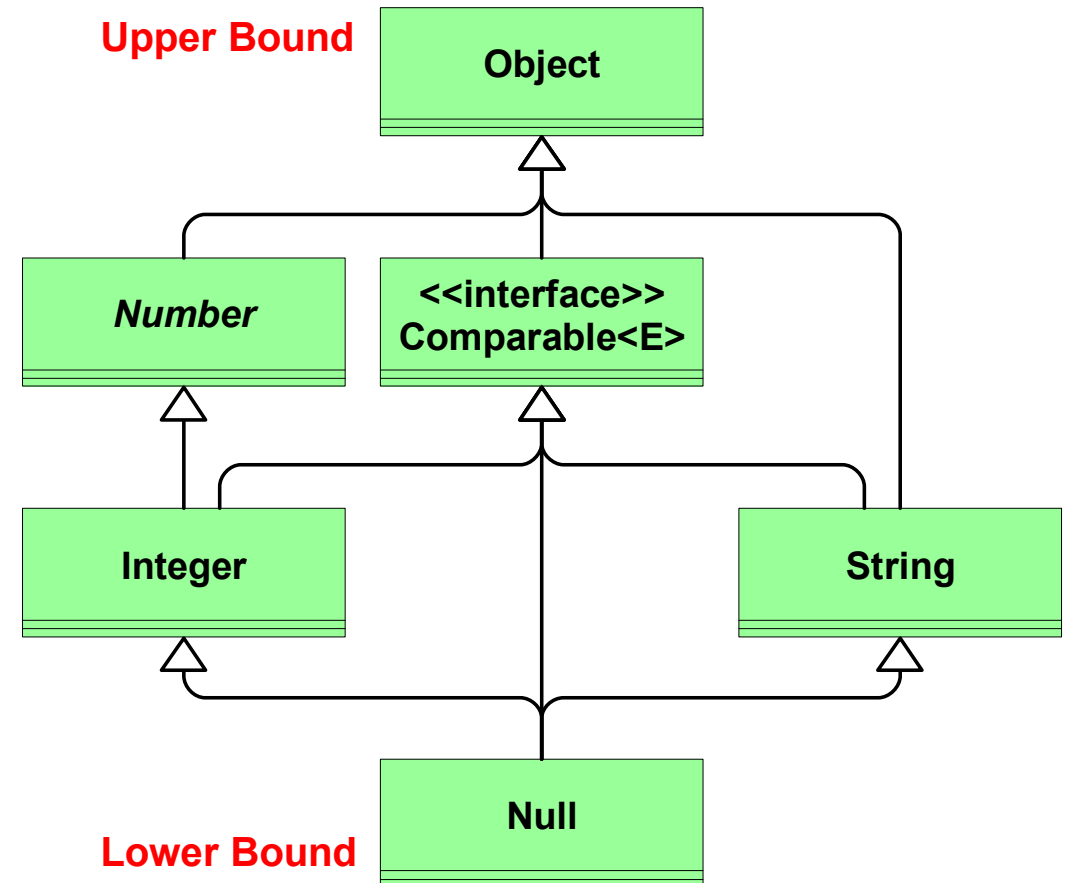
Untere Schranke für ?
Null-Typ

Beispiel-Untertypen für `List<?>`

```
List<Integer>
```

```
List<Number>
```

```
List<Object>
```



▪ Eigenschaften von Ausdrücken / Variablen des Typs `List<?>`

```
public static void printFirst(List<?> l) {  
    /* l.add( ??? ) */  
    /* ??? var = l.get(0) */  
}
```

- Wird als Argument für einen Typparameter `E` eine Wildcard angegeben, so tritt in den Methodensignaturen und als Typ für Variablen eine **besondere Typvariable** (wir nennen sie nachfolgend α) an die Stelle von `E` (wie ohne Wildcards der entsprechende Referenztyp, z.B. `List<String>`)

```
List<String> l:  
    boolean add(E value)      => boolean l.add(String value)  
    E get(int idx)           => String l.get(int idx)  
  
List<?> l:  
    boolean add(E value)      => boolean l.add( $\alpha$  value)  
    E get(int idx)           =>  $\alpha$  l.get(int idx)
```

- α ist wie gesagt eine **Typvariable** (wie in generischen Klassen-/Interface-Deklarationen) mit $\alpha <_1 u$ und $\alpha >_1 l$, wobei `u` die obere Schranke und `l` die untere Schranke der Wildcard darstellen

▪ Eigenschaften von Ausdrücken / Variablen des Typs `List<?>` (Forts.)

- Zuweisung eines Ausdrucks auf eine Variable des Typs α (**Wertübergabe**)
 - Ausdruck muss von einem **Untertypen** der unteren Schranke $\underline{1}$ von α sein – das ist ausschließlich für die Null-Referenz der Fall

```
List<?> l = new ArrayList<Integer>();  
1: l.add(null);                               /* boolean l.add( $\alpha$  value) */  
2: l.add(new Integer(42));
```

- Zuweisung eines Ausdrucks vom Typ α auf eine Variable (**Wertentnahme**)
 - Variable muss von einem **Obertypen** der oberen Schranke \overline{u} von α sein – das ist ausschließlich für `Object` der Fall

```
List<?> l = new ArrayList<Integer>();  
1: Object o = l.get(0);                         /*  $\alpha$  l.get(int idx) */  
2: System.out.println( l.get(0).toString() );  
3: Integer i = l.get(0);
```

- Welchen Typen entspricht eine Wildcard als Typparameter? (Forts.)
 - Schranken zum Typparameter des generischen Typen **schränken die Wildcard ein**

- Beispiel:**

Klasse `SortedList<E extends Comparable<E>>`

Variablendeklaration

`SortedList<?> l;`

Obere Schranke für ?

`Comparable<?>`

Untere Schranke für ?

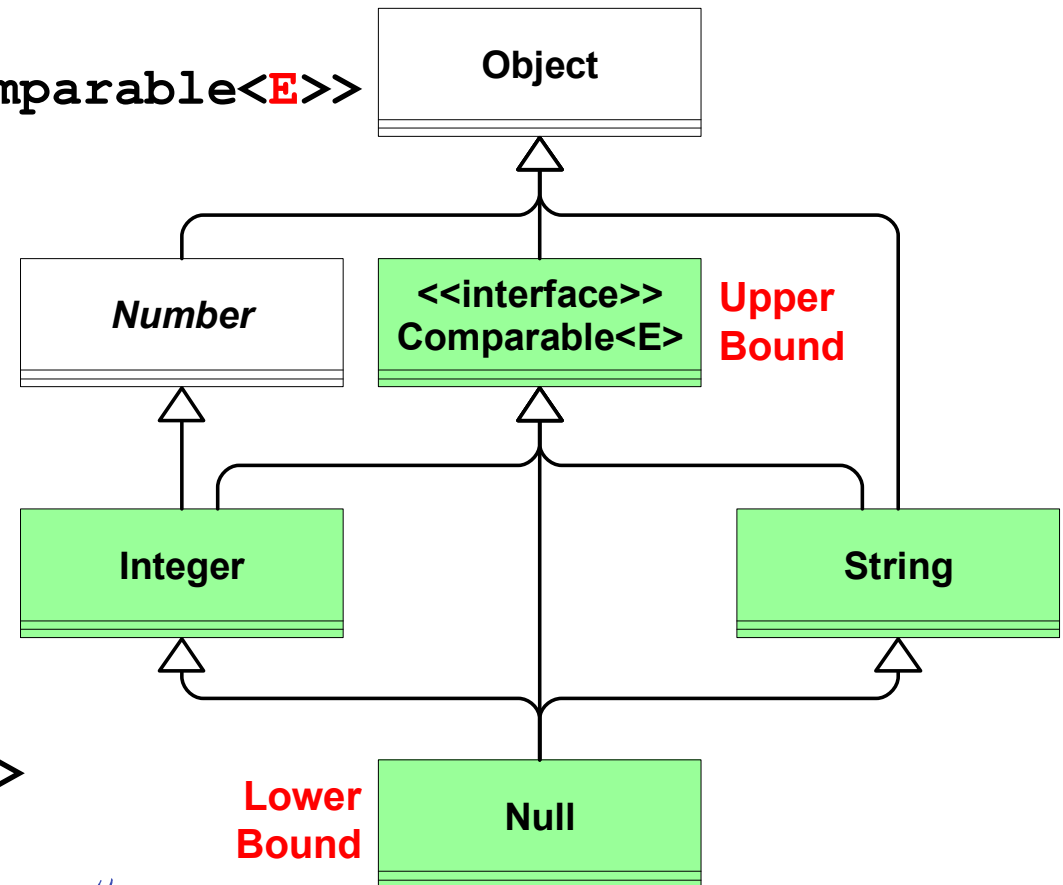
`Null-Typ`

Beispiel-Untertypen für `SortedList<?>`

`SortedList<Integer>`

`SortedList<Number>` ← implementiert comparable nicht

`SortedList<Object>`



- **Eigenschaften von Ausdrücken / Variablen des Typs `SortedList<?>`**
 - Zuweisung eines Ausdrucks auf eine Variable des Typs α (**Wertübergabe**)
 - Unverändert nur Zuweisung der **Null-Referenz**, da Null-Typ untere Schranke \perp von α
 - Zuweisung eines Ausdrucks vom Typ α auf eine Variable (**Wertentnahme**)
 - Variable muss von einem **Obertypen** einer oberen Schranke u von α sein – hier `Comparable<?>`

```
/* mit SortedList<E extends Comparable<E>> */

SortedList<?>    l = new SortedList<Integer>();

1: Object        o = l.get(0);                      /*  $\alpha$  l.get(int idx) */
2: Comparable<?> o = l.get(0);
3: String        s = l.get(0);
4: Integer        i = l.get(0);
5: System.out.println( l.get(0).toString() );
6: System.out.println( l.get(0).compareTo( ... ) );  /* Sicher? */
/* Signatur von compareTo hier: int compareTo(? o) => nur null zuweisbar */
```

■ Wildcards vs Kovarianz & Kontravarianz

- Eine Wildcard `?` beseitigt die kritischen Fälle von Kovarianz (Übergabe eines allgemeineren Wertes als der Null-Referenz) und Kontravarianz (Entnahme eines spezielleren Wertes als `Object`)
- Indem eine Wildcard nur schwache Garantien zum tatsächlichen Typen macht, werden die problematischen Operationen vom Typsystem als unsicher erkannt und sichere Operationen zugelassen
- Die Nutzung von Wildcards geschieht **individuell für einzelne Typparameter**, d.h. für jeden Typparameter im Rahmen eines parametrisierten Typen kann individuell angegeben werden, ob er beliebig ist oder nicht:

```
public class Pair<EL, ER> {  
    /* ... */  
}
```

```
Pair<?, ?>          p1 = new Pair<Integer, String>(42, "Die Antwort");    /* OK */  
Pair<?, String>     p2 = new Pair<Integer, String>(42, "Die Antwort");    /* OK */  
Pair<Integer, ?>    p3 = new Pair<Integer, String>(42, "Die Antwort");    /* OK */  
Pair<String, ?>    p4 = new Pair<Integer, String>(42, "Die Antwort");    /* Fehler */
```

- In manchen Fällen ist `object` als obere Schranke für `?` zu allgemein:

```
public static int sum(List<?> l) {  
    int result = 0;  
    for(int i = 0 ; i < l.length() ; i++)  
        result += l.get(i).intValue(); /* Compilerfehler */  
    return result;  
}  
/* ... */  
sum( new ArrayList<Number>() );      /* OK */  
sum( new ArrayList<Integer>() );     /* OK */  
sum( new ArrayList<String>() );      /* Passt zu List<?>, ist aber Unsinn */
```

- **Problem:** Umwandlung in Ganzzahl per `intValue()` nicht für beliebige Objekte (Klasse `object`) definiert
- **Lösung: Bounded wildcards**, d.h. **explizite** Festlegung der oberen Schranke (**upper bound**) des Typarguments `?`.
Auch bei Wildcards ist es möglich, den **allgemeinsten** zulässigen Typen einzuschränken und damit die implizite obere Schranke (`object`) durch eine explizite obere Schranke zu ersetzen.

```
public static int sum(List<? extends Number> l) {
    int result = 0;
    for(int i = 0 ; i < l.length() ; i++)
        result += l.get(i).intValue();    /* OK: garantierter Typ mindestens Number */
    return result;
}
/* ... */
sum( new ArrayList<Number>() );    /* OK */
sum( new ArrayList<Integer>() );    /* OK: Integer ist Untertyp von Number */
sum( new ArrayList<String>() );    /* Fehler: String ist kein Untertyp von Number */
```

- Die Wildcard `? extends Number` akzeptiert alle Typen, die Untertypen von `Number` sind (einschließlich `Number` selbst).

`sum()` kann also nur noch mit parametrisierten Listen aufgerufen werden, deren Typ-Argument `Number` oder spezieller ist (intuitiv also nur Listen von Zahltypen).

▪ Welchen Typen entspricht eine Wildcard als Typargument? (Forts.)

- Explizite Angabe der **oberen** Schranke zur Wildcard

- **Beispiel:**

Interface List<E>

Variablendeklaration

List<? extends Number> l; **Upper Bound**

Obere Schranke für ?
Number

Untere Schranke für ?
Null-Typ

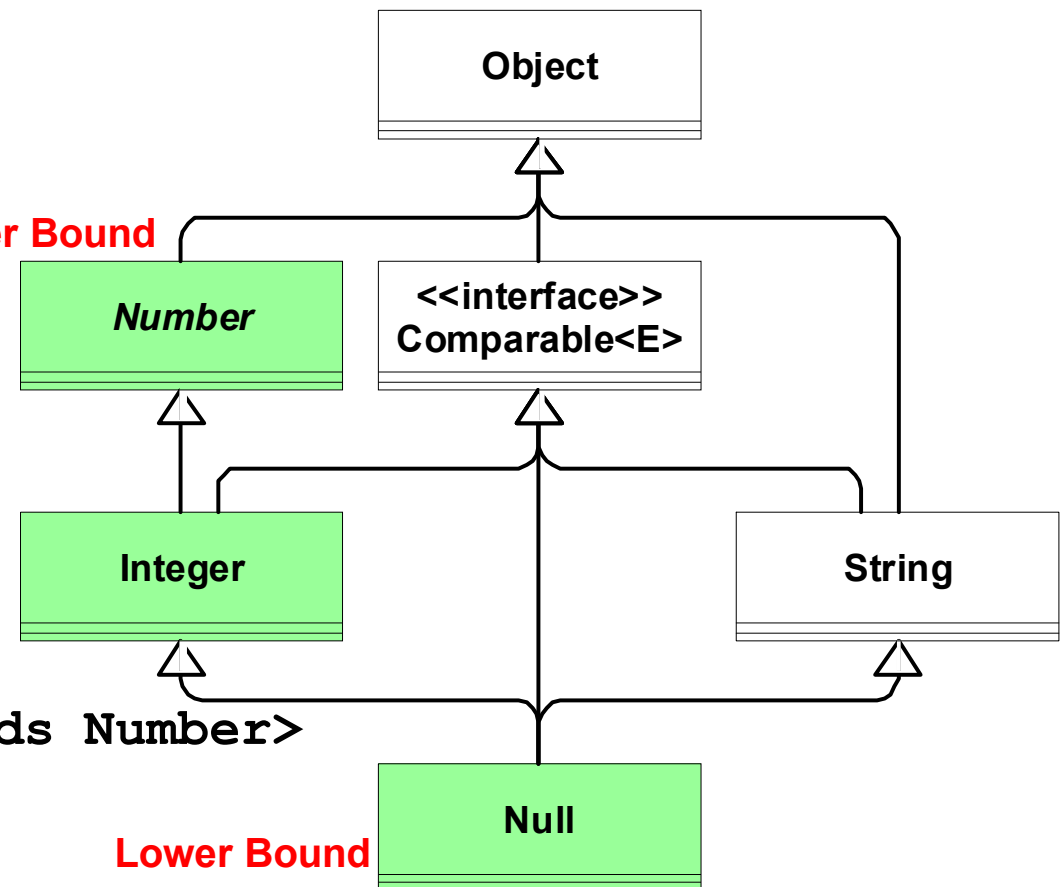
Beispiel-Untertypen für **List<? extends Number>**

List<Integer>

List<Number>

~~**List<Object>**~~

~~**List<String>**~~



Welchen Typen entspricht eine Wildcard als Typargument? (Forts.)

- Explizite Angabe der **oberen** Schranke zur Wildcard, **zusätzlich** zum Typparameter

- Beispiel:**

Klasse `SortedList<E extends Comparable<E>>`

Variablendeklaration

`SortedList<? extends Number> l;`

Obere Schranke für ?

`Comparable<?>` und `Number`

Untere Schranke für ?

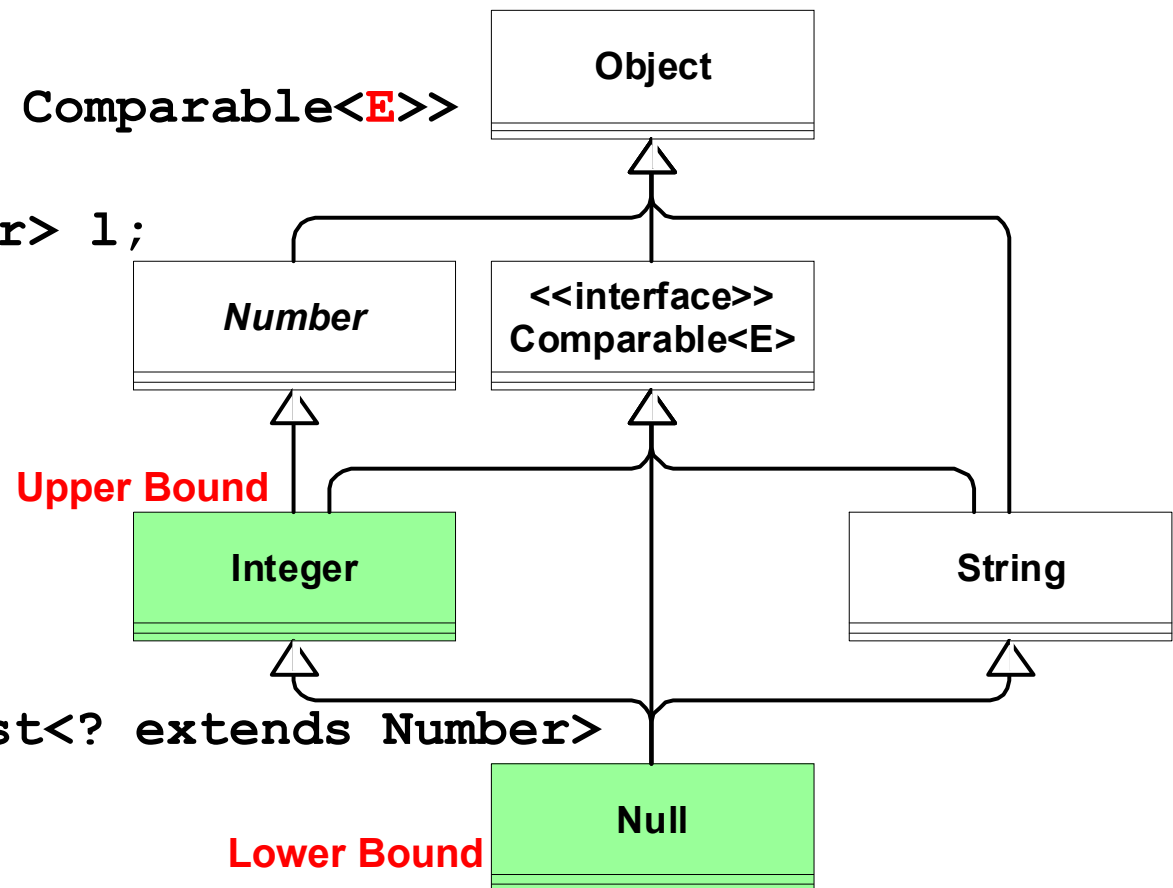
Null-Typ

Beispiel-Untertypen für `SortedList<? extends Number>`

`SortedList<Integer>`

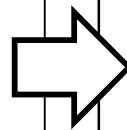
`SortedList<Number>`

`SortedList<Object>`



- Erweiterung der generischen Klasse `Pair`:

```
public class Pair<E> {  
    private E left, right;  
  
    public Pair(E left, E right) {  
        this.left = left;  
        this.right = right;  
    }  
  
    public Pair(Pair<E> other) {  
        this(other.left, other.right);  
    }  
}
```



```
public Pair(Pair<? extends E> other) {  
    this(other.left, other.right);  
}
```

```
Pair<Integer> pi = new Pair<Integer>(23,42);  
Pair<Number> pn = new Pair<Number>(pi); /* Fehler mit Pair(Pair<E>) */  
Pair<Number> pn = new Pair<Number>(pi); /* OK mit Pair(Pair<? extends E>) */
```

- Zusätzlicher Konstruktor zum Kopieren aus einem bestehenden `Pair`-Objekt
- Flexibler:** Als Elementtyp des Quell-Objektes sollte nicht nur `E`, sondern jeder Typ zulässig sein, der Untertyp von `E` (und damit zuweisungskompatibel) ist

- Nach wie vor haben wir mit Wildcards keine Möglichkeit, etwas anderes als die Null-Referenz an die Variablen eines parametrisierten Typen zu übergeben:

```
public static void sumToList(List<? extends Number> l, Number[] nums) {  
    int sum = 0;  
    for(int i = 0 ; i < nums.length ; i++)  
        sum += nums[i].intValue();  
    l.add(sum);                /* Compilerfehler, kein Untertyp von Null */  
}  
/* ... */  
sumToList( new ArrayList<Integer>(), new Number[] { 1, 2, 3 } );  
sumToList( new ArrayList<Double>(), new Number[] { 1, 2, 3 } );    /* Fehler! */
```

- Die Methode `sumToList()` soll die Summe aller Elemente eines `Number`-Arrays bilden und in eine als Parameter übergebene Liste einfügen
- Das Summieren der Array-Elemente klappt problemlos
- **Das Einfügen in die Liste hingegen scheitert:** Der tatsächliche Typ der Liste könnte z.B. `List<Double>` sein und das Einfügen eines `Integer`-Wertes (`sum`) wäre damit unsicher

- **Lösung:** Anders als für Typvariablen kann für Wildcards auch die untere Schranke (**lower bound**) explizit definiert werden, d.h. der **speziellste** zulässige Typ wird eingeschränkt und die implizite untere Schranke (Null-Typ) durch eine explizite **untere Schranke** ersetzt

```
public static void sumToList(List<? super Integer> l, Number[] nums) {
    int sum = 0;
    for(int i = 0 ; i < nums.length ; i++)
        sum += nums[i].intValue();
    l.add(sum);          /* OK */
}
/* ... */
sumToList( new ArrayList<Object>(), new Number[] { 1, 2, 3 }); /* OK */
sumToList( new ArrayList<Number>(), new Number[] { 1, 2, 3 }); /* OK */
sumToList( new ArrayList<Integer>(), new Number[] { 1, 2, 3 }); /* OK */
sumToList( new ArrayList<Double>(), new Number[] { 1, 2, 3 });
/* Fehler: Double ist kein Obertyp von Integer */
```

- Das Einfügen in die Liste ist nun möglich: Für die Wildcard der Liste kommen nur Typen in Frage, die Obertypen von `Integer` sind, d.h. jeder enthaltene Typ ist **Obertyp** von `Integer`. Ein `Integer`-Wert als (unechter) **Untertyp** von `Integer` kann daher sicher in jede hier zulässige Liste eingefügt werden.

▪ Welchen Typen entspricht eine Wildcard als Typargument? (Forts.)

- Explizite Angabe der **unteren** Schranke zur Wildcard

- **Beispiel:**

Interface List<E>

Variablendeklaration

List<? super Integer> l;

Obere Schranke für ?
Object

Untere Schranke für ?
Integer

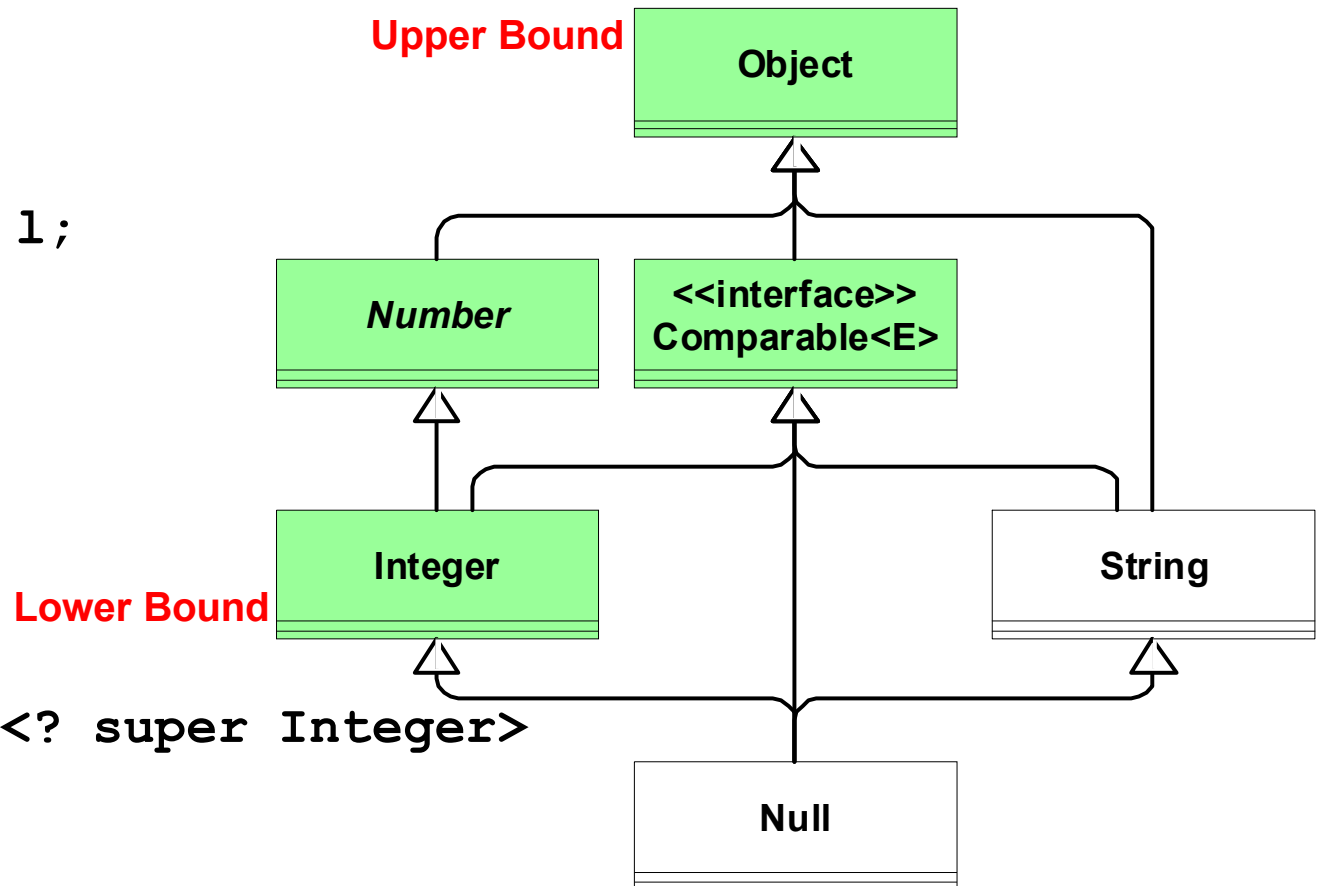
Beispiel-Untertypen für List<? super Integer>

List<Integer>

List<Number>

List<Object>

List<String>



▪ Welchen Typen entspricht eine Wildcard als Typargument? (Forts.)

- Explizite Angabe der **unteren** Schranke zur Wildcard, **zusätzlich** zum Typparameter

- **Beispiel:**

Klasse `SortedList<E extends Comparable<E>>`

Variablendeklaration

`SortedList<? super Integer> l;`

Obere Schranke für ?

`Comparable<?>`

Untere Schranke für ?

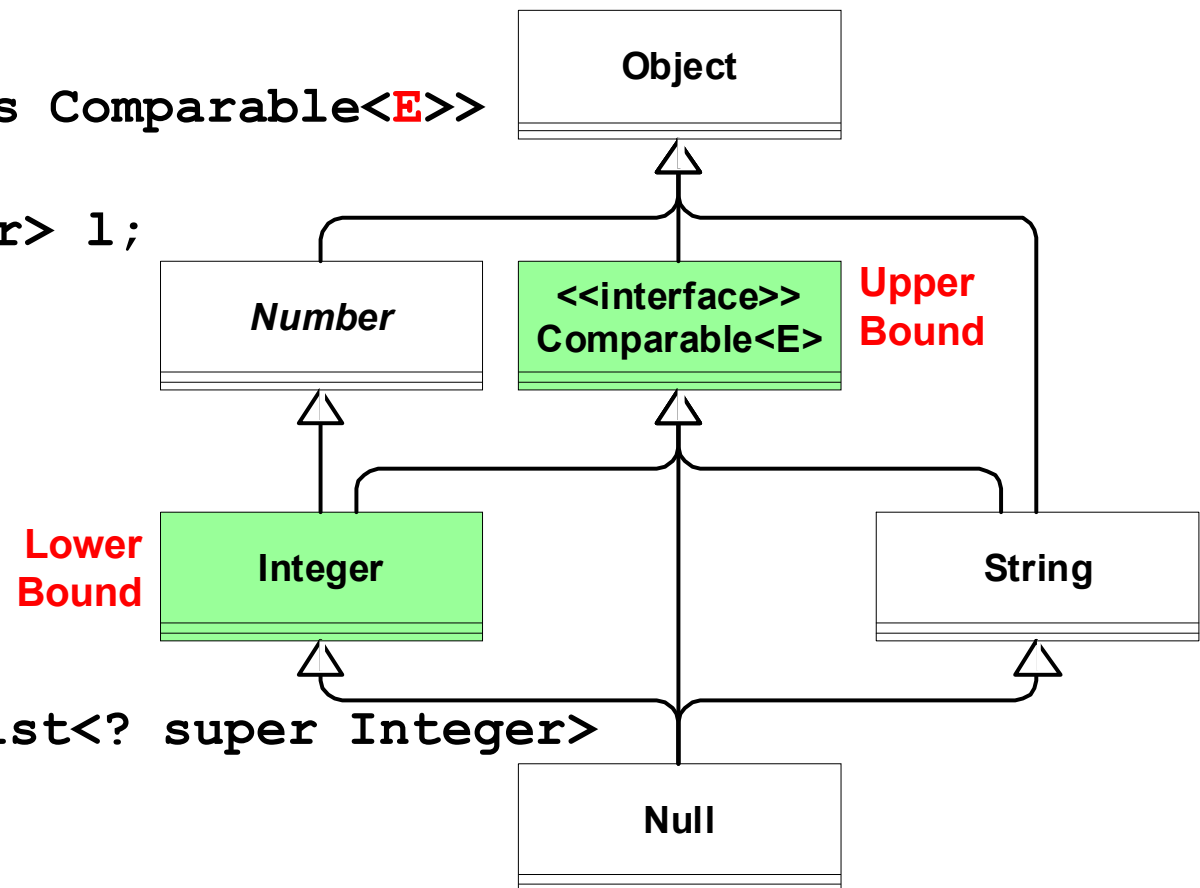
`Integer`

Beispiel-Untertypen für `SortedList<? super Integer>`

`SortedList<Integer>`

`SortedList<Number>`

`SortedList<Object>`



▪ Welchen Typen entspricht eine Wildcard als Typargument? (Forts.)

- Explizite Angabe der **unteren** Schranke zur Wildcard, **zusätzlich** zum Typparameter

- **Beispiel:**

Klasse `SortedList<E extends Comparable<E>>`

Variablendeklaration

`SortedList<? super Number> l;`

Obere Schranke für ?

`Comparable<?>`

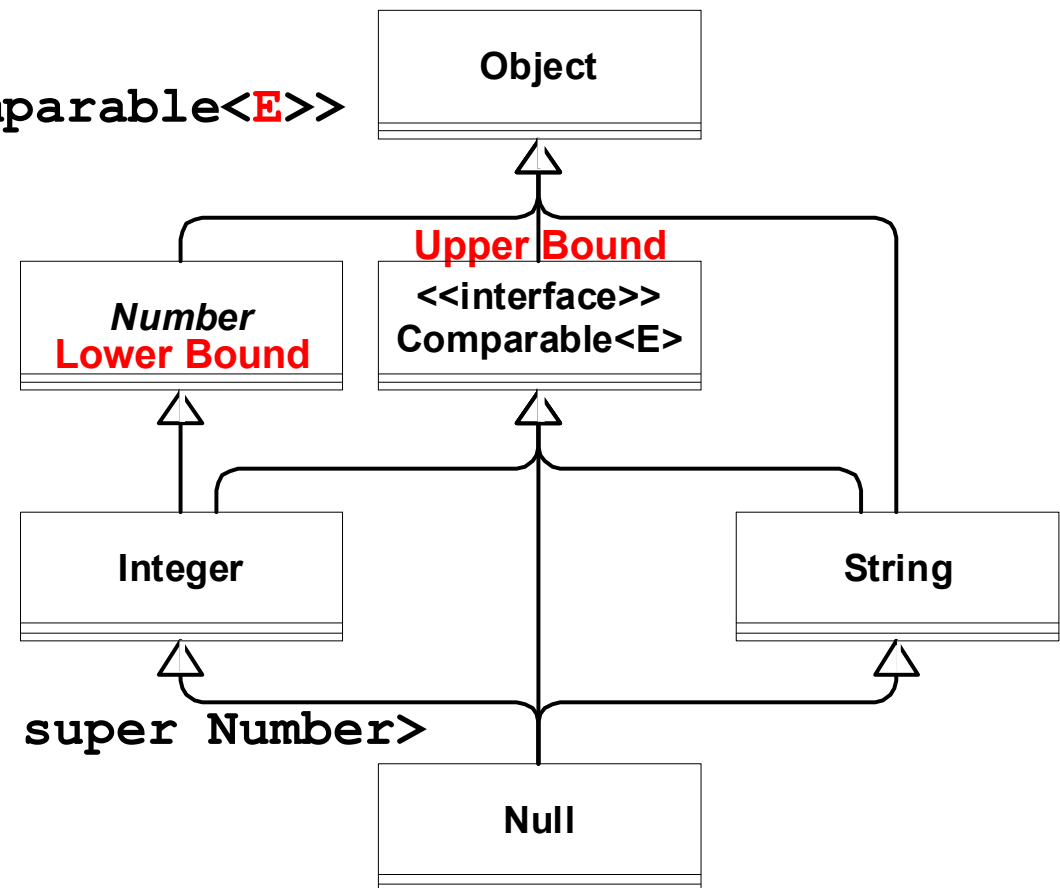
Untere Schranke für ?

`Number`

Beispiel-Untertypen für `SortedList<? super Number>`

Keine, Compiler-Fehler

Es existiert kein Referenztyp, der Obertyp von `Number` und Untertyp von `Comparable<?>` ist



▪ Kompatibilität bei **einzelnen Referenztypen** als Typparameter

```
List<Integer> l = new ArrayList<Integer>();      /* OK: identischer Typ */  
List<String> l = new ArrayList<Integer>();    /* Fehler: Typen nicht kompatibel */  
List<Number> l = new ArrayList<Integer>();    /* Fehler: Kovarianz */  
List<Integer> l = new ArrayList<Number>();    /* Fehler: Kontravarianz */
```

▪ Kompatibilität bei **Mengen von Referenztypen (Wildcards)** als Typparameter

▪ Kovariante Zuweisungen mit ? extends

```
List<? extends Number> l = new ArrayList<Number>(); /* OK: Number :> Number */  
List<? extends Number> l = new ArrayList<Integer>(); /* OK: Number :> Integer */  
Number n = l.get(0); /* Kovarianz: Wertentnahmen sind sicher */  
n.add( new Integer(42) ); /* Kovarianz: Wertübergaben sind nicht sicher */
```

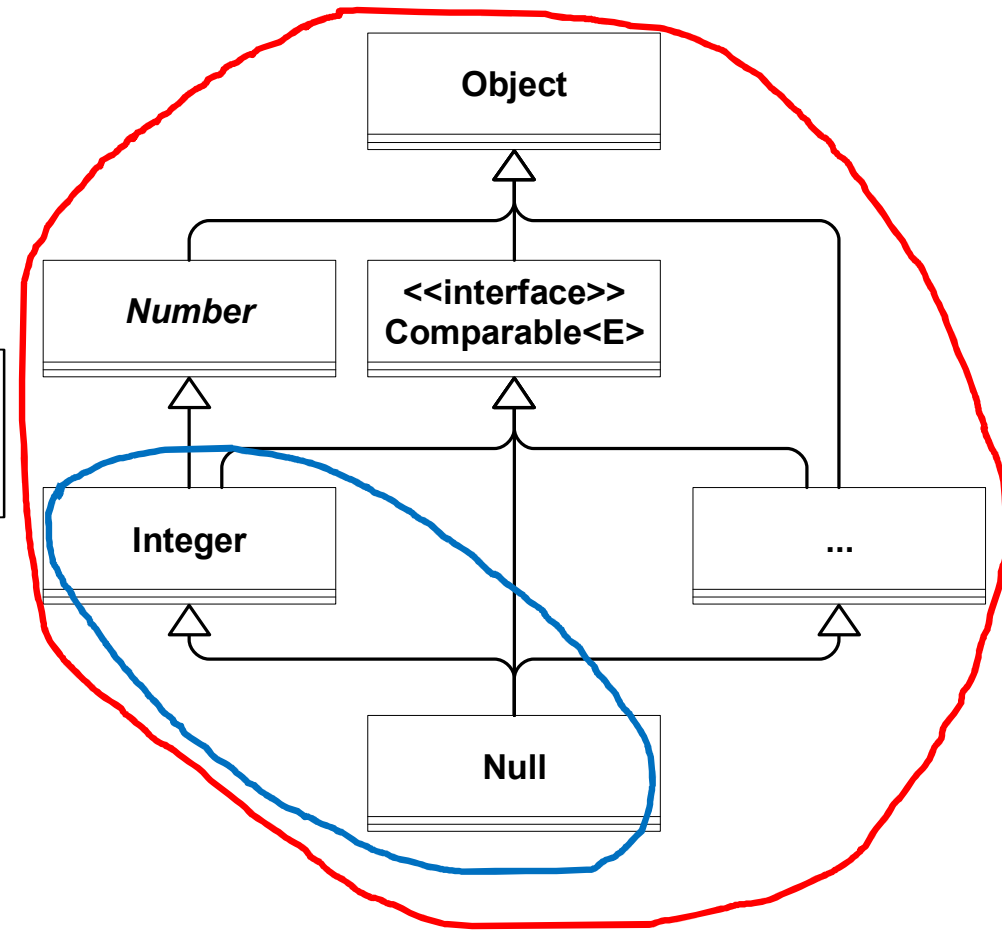
▪ Kontravariante Zuweisungen mit ? super

```
List<? super Integer> l = new ArrayList<Integer>(); /* OK: Integer <: Integer */  
List<? super Integer> l = new ArrayList<Number>(); /* OK: Integer <: Number */  
Integer i = l.get(0); /* Kontravarianz: Wertentnahmen sind nicht sicher */  
n.add( new Integer(42) ); /* Kontravarianz: Wertübergaben sind sicher */
```

- Zuweisungen zwischen parametrisierten Typen mit Wildcard
 - **Kriterium:** Enthält die Typmenge, auf die zugewiesen wird (**linke Seite**) alle Typen, von denen zugewiesen wird (**rechte Seite**)?

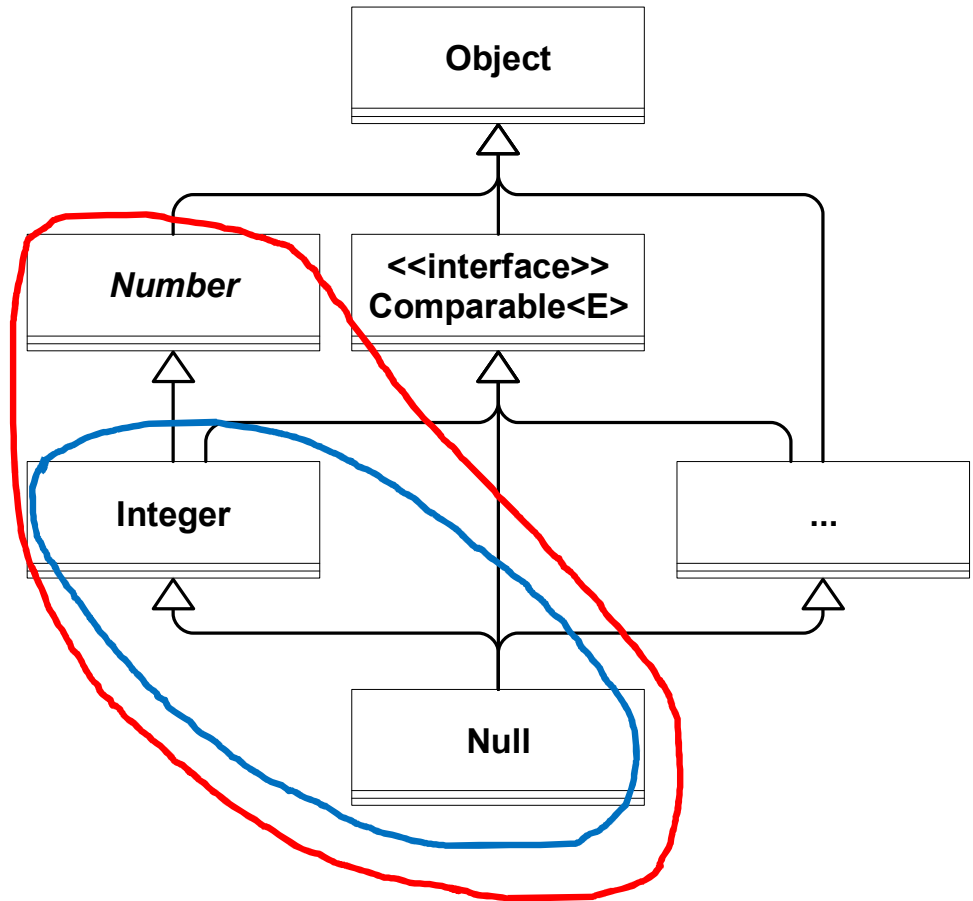
```
List<? extends Integer> xi = ...;
```

```
List<?> l = xi; /* OK */
```



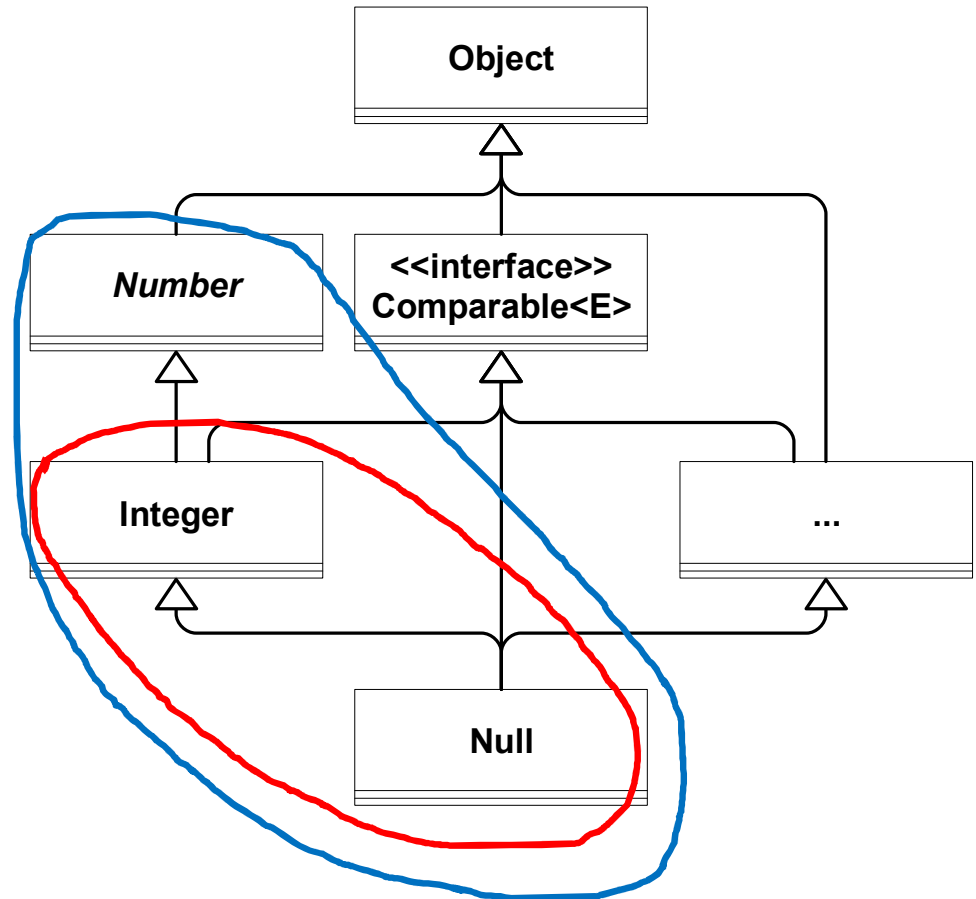
- Zuweisungen zwischen parametrisierten Typen mit Wildcard
 - **Kriterium:** Enthält die Typmenge, auf die zugewiesen wird (**linke Seite**) alle Typen, von denen zugewiesen wird (**rechte Seite**)?

```
List<? extends Integer> xi = ...;  
List<? extends Number> l = xi; /* OK */
```



- Zuweisungen zwischen parametrisierten Typen mit Wildcard
 - **Kriterium:** Enthält die Typmenge, auf die zugewiesen wird (**linke Seite**) alle Typen, von denen zugewiesen wird (**rechte Seite**)?

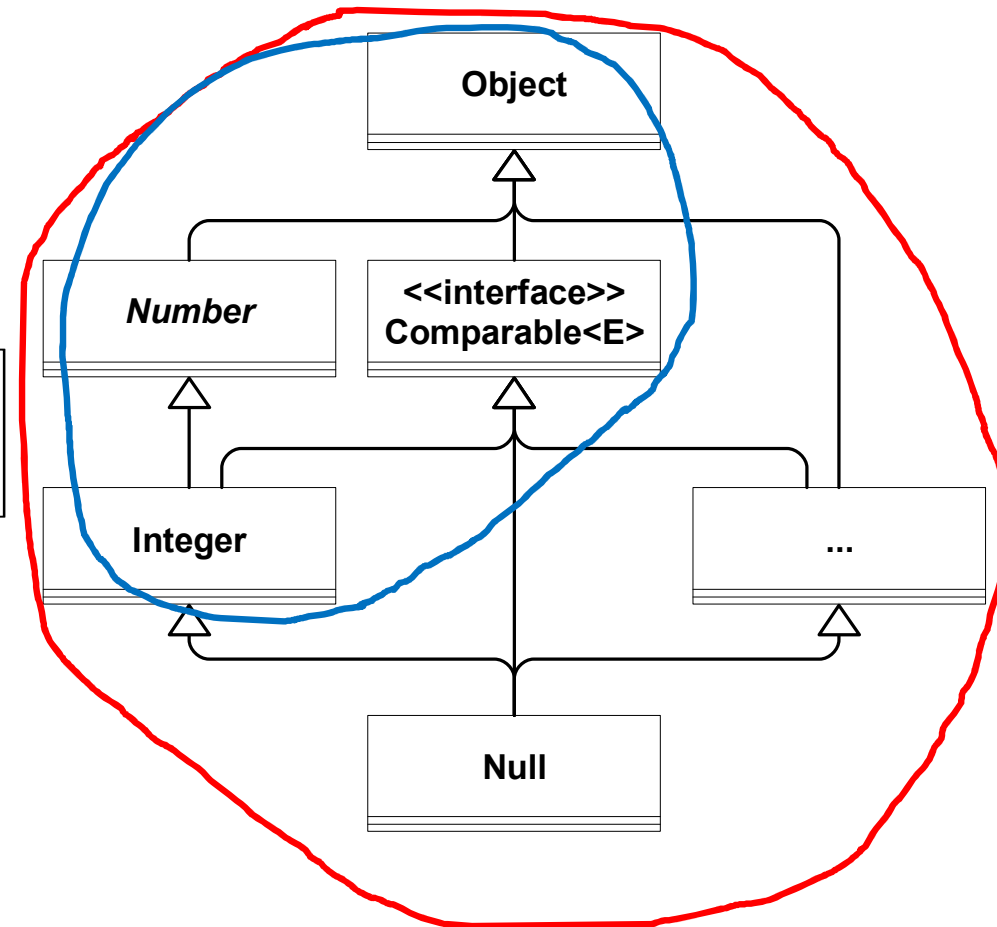
```
List<? extends Number> xn = ...;  
List<? extends Integer> l = xn; /* Fehler */
```



- Zuweisungen zwischen parametrisierten Typen mit Wildcard
 - **Kriterium:** Enthält die Typmenge, auf die zugewiesen wird (**linke Seite**) alle Typen, von denen zugewiesen wird (**rechte Seite**)?

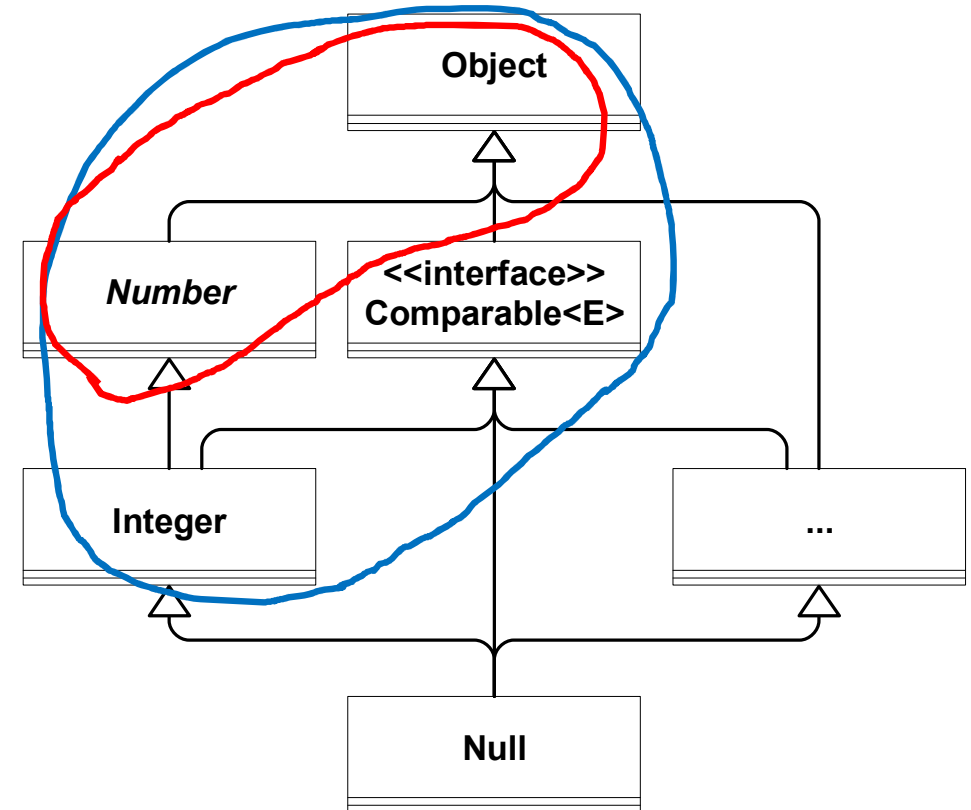
```
List<? super Integer> si = ...;
```

```
List<?> l = si; /* OK */
```



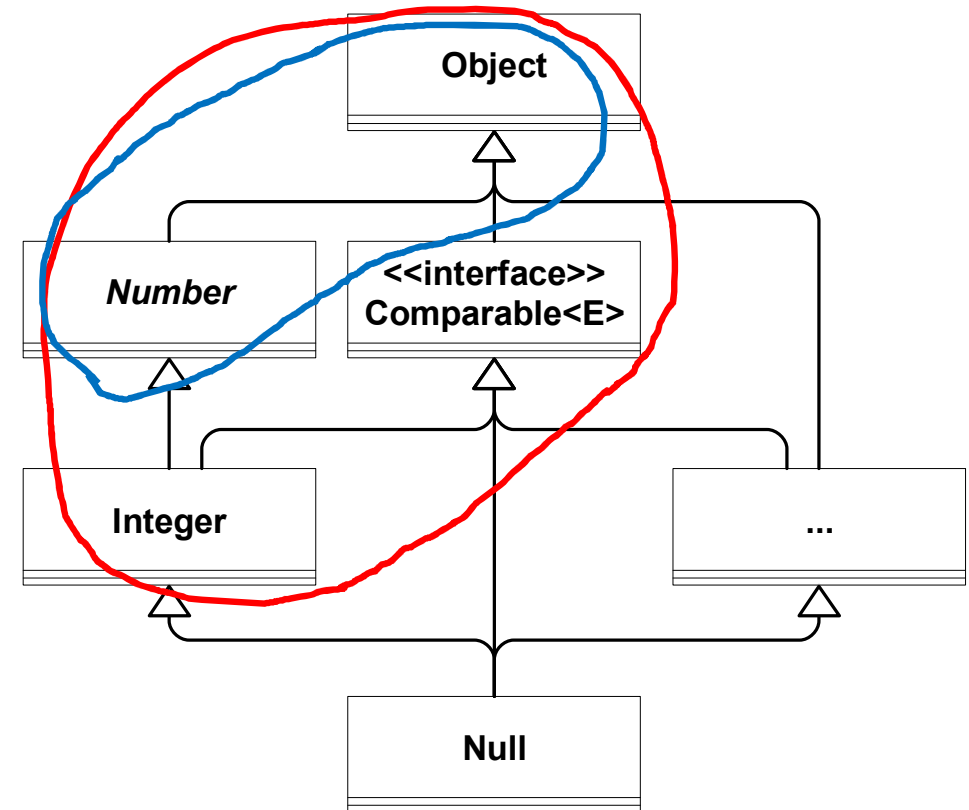
- Zuweisungen zwischen parametrisierten Typen mit Wildcard
 - **Kriterium:** Enthält die Typmenge, auf die zugewiesen wird (**linke Seite**) alle Typen, von denen zugewiesen wird (**rechte Seite**)?

```
List<? super Integer> si = ...;  
List<? super Number> l = si; /* Fehler */
```



- Zuweisungen zwischen parametrisierten Typen mit Wildcard
 - **Kriterium:** Enthält die Typmenge, auf die zugewiesen wird (**linke Seite**) alle Typen, von denen zugewiesen wird (**rechte Seite**)?

```
List<? super Number> sn = ...;  
List<? super Integer> l = sn; /* OK */
```



Identifizier *[TypeArguments]*

- **Identifizier**: Bezeichner eines **generischen Klassen- oder Interface-Typen**
- **TypeArguments**: Angaben eines Typarguments zu jedem Typparameter des generischen Typen, wobei ein Typargument entweder ein Referenztyp oder eine Wildcard ist
 - $\text{TypeArguments} = \langle (\text{ReferenceType} \mid \text{Wildcard}) \{, (\text{ReferenceType} \mid \text{Wildcard}) \} \rangle$
- Referenztypen sind: Klassen-Typen, Interface-Typen, Typvariablen, Array-Typen
- **Wildcard**: Angabe einer **Menge von Referenztypen** für einen Typparameter, dabei **optional** Angabe **genau einer** oberen Schranke oder **genau einer** unteren Schranke
 - ? *entspricht ? extends Object*
 - ? **extends** *ReferenceType*
 - ? **super** *ReferenceType*
- **Beispiele**
 - `List<? super Number>`
 - `Pair<Customer, List<? extends Receipt>>`

- Ein parametrisierter Typ `c<T1, ..., Tn>` muss außerdem folgenden Anforderungen genügen:
 - Die **Anzahl Typargumente** entspricht der Anzahl deklarerter formaler Typparameter des generischen Typen `c`. Für `List<E>` wäre damit `List<String>` eine gültige Parametrisierung, `List<String, Integer>` hingegen nicht.
 - Ein als Typargument angegebener **Referenztyp** muss Untertyp **aller** oberen Schranken des Typparameters sein

```
public class Compute<E extends Number> { /* ... */ }  
Compute<String> cs;          /* Fehler: String kein Untertyp von Number */
```

- Wird zu einer **Wildcard** eine untere Schranke angegeben, so muss diese Untertyp zu allen oberen Schranken des Typparameters sein (**sonst: leere Typmenge**)

```
Compute<? super String> cs; /* Fehler: String kein Untertyp von Number */
```

- Zu einer **Wildcard** dürfen nie mehrere Schranken bzw. nie gleichzeitig eine obere und eine untere Schranke angegeben werden

- Grundsätzlich darf ein parametrisierter Typ überall eingesetzt werden, wo ein Referenztyp erwartet wird
- **Ausnahme:** Parametrisierte Typen sind in der `throws`-Klausel einer Methode / eines Konstruktors sowie als Parameter eines `catch`-Blocks unzulässig
- **Ausnahme:** An manchen Stellen sind parametrisierte Typen zwar zulässig, dürfen aber keine Wildcards enthalten
 - **Oberklassen bzw. -interfaces** in Klassen- bzw. Interface-Deklarationen (`extends`, `implements`)

```
public class CleverList implements List<? extends Number> {           /* Fehler! */
    /* ... */
}
```

- Zu **instanzierender Typ** im Rahmen eines Konstruktoraufrufs

```
public class Stuff<E> {
    List<?> l = new ArrayList<?>(); // Fehler!
    List<? extends Number> ln = new ArrayList<? extends Number>(); // Fehler!
}
```

„legend/wann muss man sich festlegen.“

- **Beispiel:** Elemente eines Arrays in eine Liste kopieren

```
public static void copy(Object[] arr, List<?> l) {  
    for(int i = 0 ; i < arr.length ; i++)  
        l.add(arr[i]); /* Compilerfehler */  
}
```

- Gleiches Problem wie zuvor:

- **Upper bound** von ? in `List<?>` ist `object`, die **lower bound** ist der Null-Typ, die Elemente des Arrays `arr` haben den Übersetzungszeit-Typen `object`
- Dem Parameter `value` von `l.add(α value)` kann also in gesicherter Weise nur die Null-Referenz übergeben werden.

- **Beispiel für Konflikt:**

```
Object[]  objs = new Object[] { "Zweiundvierzig", "Leerer String" };  
List<Integer> l = new ArrayList<Integer>();  
copy(objs, l); /* Strings gehören nicht in eine Integer-Liste */
```

- Der Typ `List<Integer>` garantiert, dass jedes Element zu `Integer` kompatibel ist (auch wenn es zur Laufzeit spezieller sein mag). Die Elemente von `objs` garantieren lediglich (trivialerweise), dass sie alle zu `object` kompatibel sind.

- Herausforderung: Die Methode `copy()` soll...
 - ... auf **jedes Array eines Referenztypen** anwendbar sein
 - ... typsicher bzgl. der generischen Zielliste sein, d.h. die übergebene Liste muss einen **zum Array passenden Typ** aufweisen
- **Lösung:** Einführung von Typparametern **für einzelne Methoden** (statt wie bisher Typparameter für ganze Typen, d.h. Klassen und Interfaces)

```
public static <T> void copy(T[] arr, List<T> l) {  
    for(int i = 0 ; i < arr.length ; i++)  
        l.add(arr[i]);  
}
```

- Für die Methode `copy()` wird ein Typparameter mit der Typvariablen `T` deklariert. Der Sichtbarkeitsbereich dieser Typvariablen beschränkt sich auf die Methodendeklaration.
- Über die Typvariable `T` wird zwischen dem **Typ der Liste** und dem **Typ des Arrays** ein Bezug hergestellt

```
public class Utils {  
    /* ... */  
    public static <T> void copy(T[] arr, List<T> l) {  
        for(int i = 0 ; i < arr.length ; i++)  
            l.add(arr[i]);  
    }  
    /* ... */  
}
```

- Das Array `arr` muss zu `T` zuweisungskompatible Elemente **enthalten**, die Liste `l` muss zu `T` zuweisungskompatible Elemente **aufnehmen** können
- Typargumente müssen beim Aufruf nicht angegeben werden, sie werden dann aus den tatsächlichen Methodenargumenten geschlossen (Typinferenz)
- **Aufrufbeispiele:**

```
Integer[]    is = { 1, 2, 3 };  
List<Integer> li = new ArrayList<Integer>();  
  
/* explizite Angabe der Typparameter (nur qualifizierte Methodenaufrufe) */  
Utils.<Integer>copy(is, li);  
/* Typparameter inferiert aus Array: T = Integer */  
copy(is, li);
```

- `List<T>` ist unnötig streng: Z.B. sollten die Elemente eines `Integer`-Arrays in eine `Number`-Liste übertragbar sein
- **Flexibilisierung per Wildcard:** Die untere Schranke für den Elementtyp der Liste muss dem Array-Elementtyp entsprechen. Allgemeinere Typen bis hin zu `List<Object>` sind unproblematisch:

```
public static <T> void copy(T[] arr, List<? super T> l) {  
    for(int i = 0 ; i < arr.length ; i++)  
        l.add(arr[i]);  
}
```

- **Aufrufbeispiele:**

```
Integer[] is = { 1, 2, 3 };  
  
List<Number> ln = new ArrayList<Number>();  
copy(is, ln);      /* Inferiert aus Array: T = Integer */  
List<? super Number> lo = new ArrayList<Object>();  
copy(is, lo);      /* Inferiert aus Array: T = Integer */  
List<String> ls = new ArrayList<String>();  
copy(is, ls);    /* String ist kein Obertyp von Integer */
```

*{MethodModifier} **TypeParameters** Result Identifier ([FormalParameterList]) [Throws] MethodBody*

- Methoden können eigene Typparameter deklarieren (unabhängig von bzw. gegebenenfalls zusätzlich zu Typparametern der Klasse bzw. des Interfaces)
- Der Sichtbarkeitsbereich einer Typvariable umfasst syntaktisch die gesamte Methodendeklaration (Typparameter, Rückgabewert, Parameter, Rumpf ...)
- Die Typvariablen sind insbesondere verwendbar für Parametertypen, Rückgabotyp und lokale Variablen
- Analog zu generischen Klassen können Typparameter von generischen Methoden per Schlüsselwort **extends** mit Schranken versehen werden
- Generische Methoden können in nicht-generischen Typen vorkommen, ebenso wie generische Typen keine generischen Methoden aufweisen müssen

```
public interface Transformer<S, T> {  
    T transform(S input);  
}
```

■ Beispiel: Generische Methode `apply()`

```
public class Pair<E> {  
  
    /* ... */  
  
    public <F> Pair<F> apply(Transformer<? super E, ? extends F> t) {  
        return new Pair<F>(  
            t.transform(this.l()),  
            t.transform(this.r())  
        );  
    }  
  
    public String toString() {  
        return this.l() + ":" + this.r();  
    }  
  
}
```


- **Beispiel:** Anwendung der generischen Methode `apply()` mit einer `Transformer`-Instanz zur Konvertierung `Double` -> `Integer`

```
public class Db1ToInt
implements Transformer<Double, Integer> {

    public Integer transform(Double input) {
        return (int) Math.round(input);
    }

}

Pair<Double> pd = new Pair<Double>(22.8, 42.2);
System.out.println(pd);
> 22.8:42.2

Pair<Integer> pi = pd.apply( new Db1ToInt() );
System.out.println(pi);
> 23:42
```

- **Beispiel:** Anwendung der generischen Methode `apply()` mit einer `Transformer`-Instanz zur Multiplikation mit einem Faktor (Konstruktorargument)

```
public class Multiply
implements Transformer<Integer, Integer> {
    private int factor;

    public Multiply(int factor) {
        this.factor = factor;
    }

    public Integer transform(Integer input) {
        return input * this.factor;
    }
}

Pair<Integer> p = new Pair<Integer>(23, 42);
System.out.println(p);
> 23:42

p = p.apply( new Multiply(3) );
System.out.println(p);
> 69:126
```

- **Beispiel:** Ein **Transformer**, der zwei **Transformer**-Instanzen typsicher zu einer Instanz zusammenfasst
 - Zur Verbindung der beiden **Transformer** ist ein Zwischentyp **I** erforderlich

```
public class Compose<S,I,T>
implements Transformer<S,T> {

    private Transformer<? super S, ? extends I> left;
    private Transformer<? super I, ? extends T> right;

    public Compose(Transformer<? super S, ? extends I> left,
                    Transformer<? super I, ? extends T> right) {
        this.left = left;
        this.right = right;
    }

    public T transform(S input) {
        return right.transform( left.transform(input) );
    }

}
```

- **Beispiel:** Transformer `ToString` zur Ausgabe eines beliebigen Objektes

```
public class ToString
implements Transformer<Object, String> {
    public String transform(Object input) {
        return input.toString();
    }
}
```

- **Beispiel:** Anwendung von `apply()` mit drei Transformer-Instanzen

```
Pair<Double> pd = new Pair<Double>(22.8, 42.2);
Pair<String> ps =
    pd.apply(
        new Compose<>(
            new Compose<>(
                new Db1ToInt(),
                new Multiply(3)
            ),
            new ToString()
        )
    );
System.out.println(ps);
> 69:126
```

- Deklaration eines **generischen Konstruktors**
 - wie bei generischen Methoden...
- Aufruf eines **generischen Konstruktors**
 - wie bei generischen Methoden

```
public class Pair<E> {  
  
    public <O> Pair(Pair<O> p, Transformer<? super O, ? extends E> t) {  
        this.left = t.transform(p.l());  
        this.right = t.transform(p.r());  
    }  
  
    /* ... */  
}  
  
Pair<Double> pd = new Pair<Double>(2.3, 4.2);  
Pair<Integer> pi = new Pair<Integer>(pd, new Db1ToInt());  
System.out.println(pi);  
> 2:4  
oder: Pair<Integer> pi = new <Double>Pair<Integer>(pd, new Db1ToInt());
```

- Arrays und Generics vertragen sich in Java nur schlecht
 - **Problem 1:** Generische Typinformationen (d.h. parametrisierte Typen) **stehen zur Laufzeit nicht zur Verfügung**
 - **Problem 2:** Anders als parametrisierte Typen sind Array-Typen in Java **kovariant**, d.h. es gilt: $x <: y \Rightarrow x[] <: y[]$

Beispiel: `Integer[]` ist ein Untertyp von `Number[]`, außerdem ist jedes Array ein Untertyp von `Object[]` (und `Object`)

- Die Kovarianz von Arrays ermöglicht Typfehler zur Laufzeit:

```
Number[] nums = new Integer[10]; /* OK: Arrays sind kovariant */
nums[0]       = new Double(4.2); /* Statisch OK, Typfehler zur Laufzeit */
Number n      = nums[0];         /* wird nie erreicht... */
```

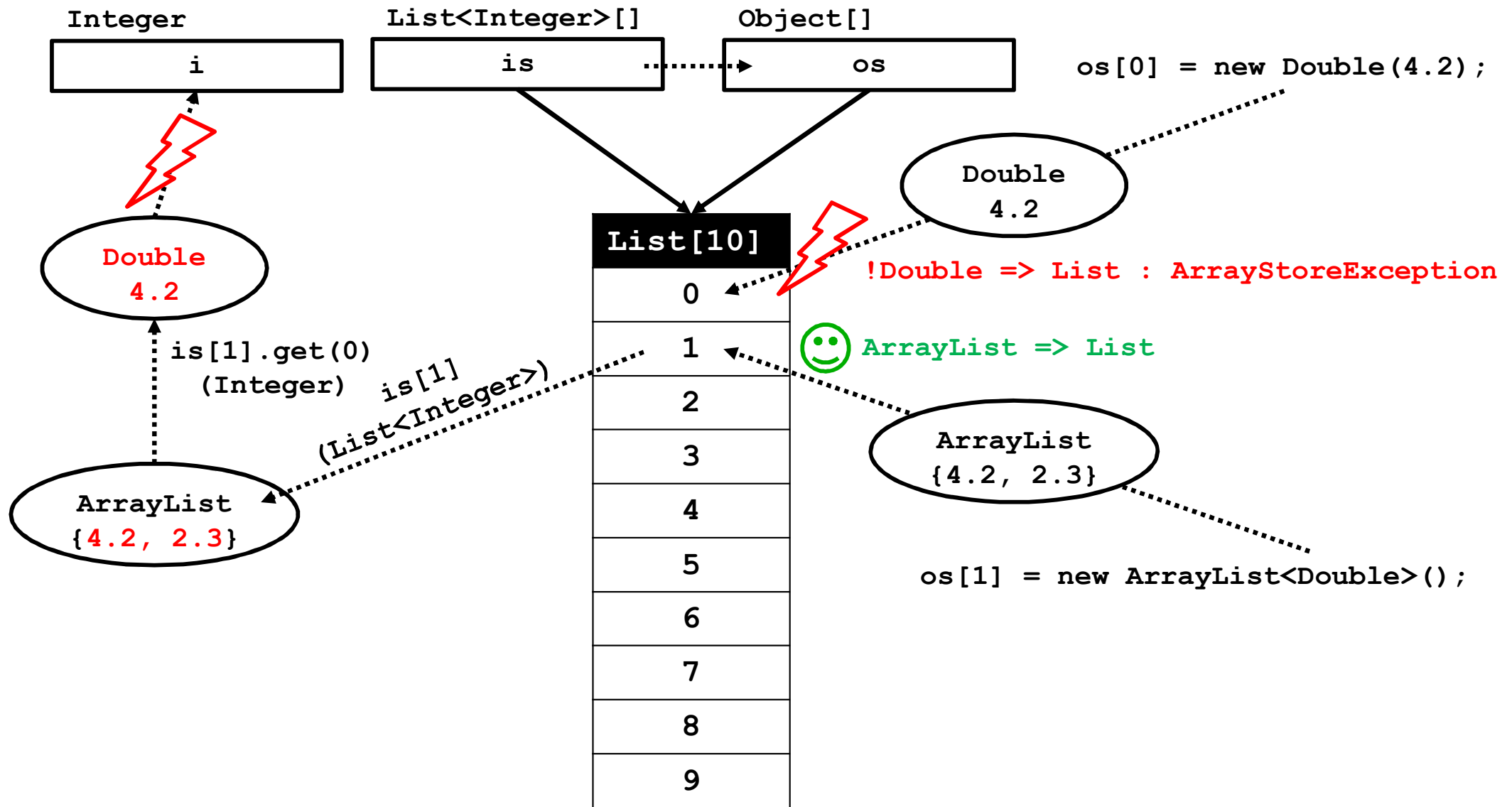
- Das Beispiel lässt sich **ohne Compilerwarnungen** übersetzen. Zur Laufzeit jedoch wird erkannt, dass in ein `Integer`-Array ein nicht zuweisungskompatibler Wert (Typ `Double`) eingefügt werden soll - es kommt zu einer `ArrayStoreException`.

- Um eine `ArrayStoreException` zu erkennen, muss zu einem Array-Objekt zur Laufzeit bekannt sein, welchen Typ die Elemente haben dürfen.

Beispiel mit **parametrisierten Typen**:

```
List<Integer>[] is = new List<Integer>[10]; /* Compilerfehler */
Object[]      os = is;                    /* OK: Arrays sind kovariant */
os[0]         = new ArrayList<Double>();
Integer i     = is[0].get(0);
```

- Da die parametrisierten Typen zur Laufzeit nur noch in ihrer nicht-parametrisierten Form (`List` bzw. `ArrayList`) existieren, könnte die für `is` unzulässige Zuweisung einer `ArrayList<Double>` nicht erkannt werden. Im Anschluss würde die Zuweisung auf die `Integer`-Variable `i` zu einem Typfehler führen müssen, da in Wahrheit ein `Double` vorliegt.
- Java Generics garantieren aber Typsicherheit, d.h. wenn zur **Übersetzungszeit** keine Typfehler oder -warnungen erkannt werden, können auch zur **Laufzeit** keine Typfehler auftreten
=> unvereinbar mit der Kovarianz von Arrays



- **Daher:** Elementtyp eines Arrays-Objektes **darf kein parametrisierter Typ sein**
- Für die Elemente eines Array-**Typen** ist ein parametrisierter Typ zulässig, nicht jedoch für die Konstruktion eines Array-**Objektes**:

```
List<Integer>[] is;           /* OK */
is = new List<Integer>[10]; /* Compilerfehler */
is = new List[10];           /* Warnung: unchecked conversion */
is = (List<Integer>[])new List[10]; /* Warnung: unchecked cast */
```

- Botschaft: Ist mit **unchecked conversion** bzw. **unchecked cast** kompilierbar, aber unsicher, d.h. zur Laufzeit können Typfehler auftreten
=> Verantwortung für sichere Verwendung liegt beim Entwickler

```
List<Integer>[] is = new List<Integer>[10]; /* Compilerfehler */
List<Integer>[] is = (List<Integer>[])new List[10]; /* Warnung: unchecked cast */
Object[]      os = is;
List<Double>   ld = new ArrayList<Double>();
ld.add(4.2);
os[0]          = ld; /* keine ArrayStoreException ! */
Integer i      = is[0].get(0); /* ClassCastException zur Laufzeit */
```

- Java Generics ist parametrische Polymorphie zur **Übersetzungszeit**
 - Die generischen Typinformationen (parametrisierte Typen) stehen zur Laufzeit nicht zur Verfügung (soweit es die virtuelle Maschine anbelangt)
 - Für eine generische Klasse wird nur eine einzige **.class**-Datei erzeugt, so als wenn die Klasse nicht generisch wäre (d.h. der Bytecode für alle parametrisierten Typen eines generischen Typen ist derselbe)
 - Folglich ist nichts möglich, was den parametrisierten Typen zur Laufzeit erfordern würde
 - Wie schon besprochen: Elementtyp von Arrays darf **nicht parametrisiert** sein (Ausnahme: Unbeschränkte Wildcards für alle Typparameter)
 - Typvariablen können nicht für Konstruktoren verwendet werden (da ein Konstruktoraufwurf in der VM erfordert, dass die tatsächlich zu instanziiierende Klasse im Bytecode bekannt ist)

```
public class Pair<A> {  
    private A left  = new A(); /* Compilerfehler */  
    private A right = new A(); /* Compilerfehler */  
}
```

- Java Generics ist parametrische Polymorphie zur **Übersetzungszeit** (Forts.)
 - Folglich ist nichts möglich, was den parametrisierten Typen zur Laufzeit erfordern würde (Forts.)
 - Gleiches gilt für die Array-Erzeugung: Es kann kein Array mit einer **Typvariablen als Elementtyp** erzeugt werden (da auch hier im Bytecode nicht bekannt wäre, von welchem Elementtyp das Array wirklich sein soll)

```
public class ArrayList<A> {  
    private A[] arr = new A[10]; /* Compilerfehler */  
}
```

- Dieser Fall ist nicht identisch mit dem Verbot parametrisierter Typen als Elementtyp - eine Typvariable kann auch für einen nicht-parametrisierten Typen stehen (z.B. **A = Integer** im Falle einer **ArrayList<Integer>**)
- Der Operator **instanceof**, der den Laufzeittyp eines Objektes prüft, ist auf parametrisierte Typen nicht anwendbar (**Ausnahme**: Unbounded Wildcards)

```
if(obj instanceof Pair<Integer>) { /* ... */ } /* Fehler */  
if(obj instanceof Pair<?>)      { /* ... */ } /* OK, Unbounded Wildcard */  
if(obj instanceof Pair)         { /* ... */ } /* OK, Raw Type */
```

- Java Generics lässt nur Referenztypen als Typen bzw. Typargumente zu, `List<int>` ist also unzulässig
- **Grund auch hier:** Die Handhabung primitiver Typen erfordert besonderen Bytecode (z.B. für Konvertierung). Während für Referenztypen einheitlicher Bytecode erzeugt werden kann, würden primitive Typen eine Fallunterscheidung erforderlich machen.
- Häufig in Hinblick auf die verfügbaren Wrapper-Klassen kein großes Problem:

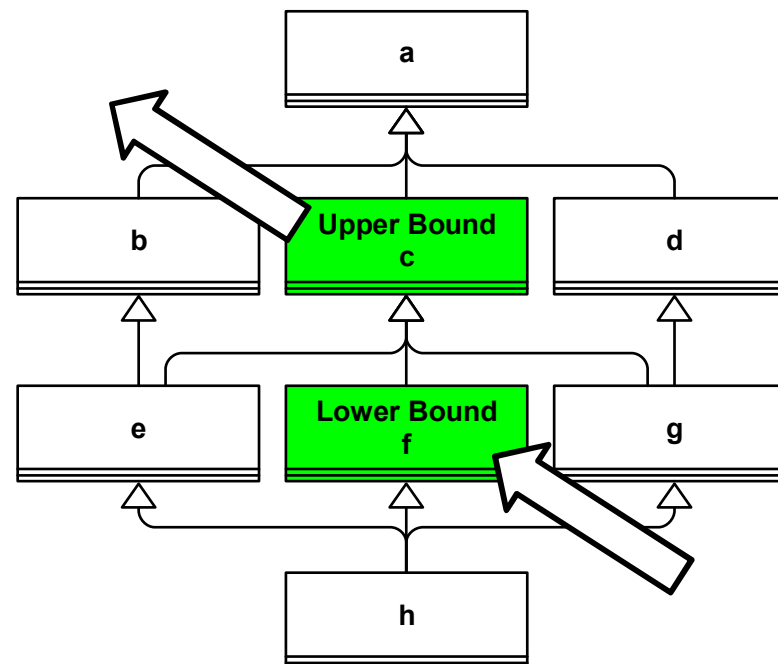
```
List<int> l1 = new ArrayList<int>();      /* Compilerfehler */
List<Integer> l2 = new ArrayList<Integer>(); /* OK */
l2.add(new Integer(42));                  /* OK */
l2.add(43);                              /* OK, Autoboxing */
Integer i1 = l2.get(0);                   /* OK */
int i2 = l2.get(0);                       /* OK, Autoboxing */
```

- **Nachteil:** Die Wrapper-Typen enthalten wie jeder Referenztyp auch die Null-Referenz als möglichen Wert
=> `int i2 = l2.get(0)` ist insofern unsicher und kann zur Laufzeit zu einer `NullPointerException` führen

- Eine Typvariable τ steht für eine bestimmte **Menge von Referenztypen**, die durch untere und obere Schranken definiert wird
- Eine Typvariable τ kann selbst als **Referenztyp** verwendet werden, d.h. Variablen bzw. Ausdrücke können vom Typ τ sein
- Zuweisungen unter Beteiligung von Variablen bzw. Ausdrücken des Typs τ sind eingeschränkt und müssen für alle Typen der Typmenge τ sicher sein

```
 $\tau$  right = ...;  
v left = right;  
v :> ub( $\tau$ )
```

Zuweisung **von** einem Ausdruck des Typs τ entspricht einer Zuweisung vom Typ der **oberen Schranke** $ub(\tau)$, es muss also auf einen Obertypen der oberen Schranke zugewiesen werden



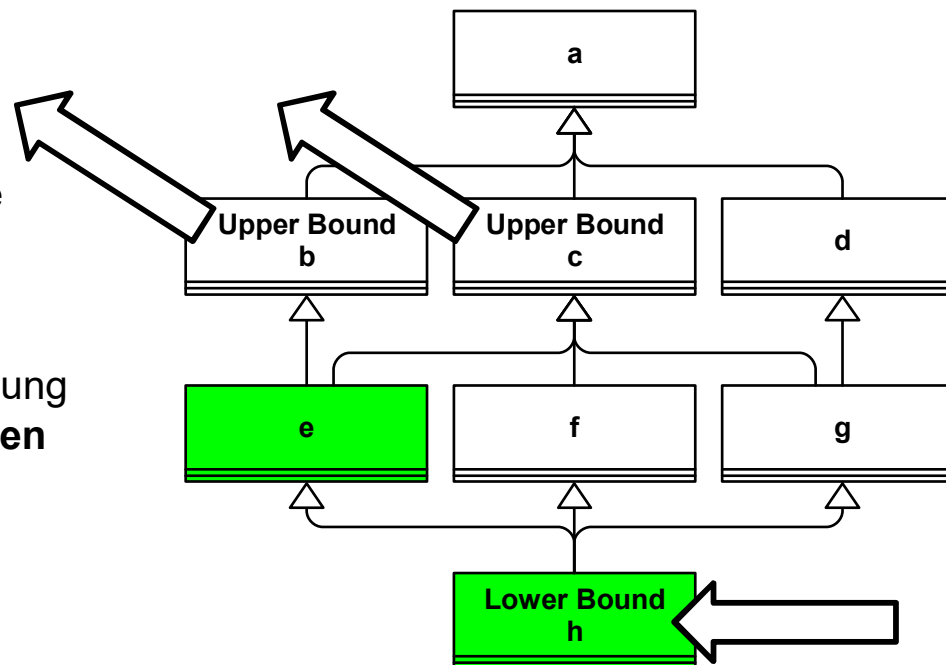
Zuweisung **auf** eine Variable vom Typ τ entspricht einer Zuweisung auf den Typ der **unteren Schranke** $lb(\tau)$, der zugewiesene Ausdruck muss also von einem Untertyp der unteren Schranke sein

```
v right = ...;  
 $\tau$  left = right;  
lb( $\tau$ ) :> v
```

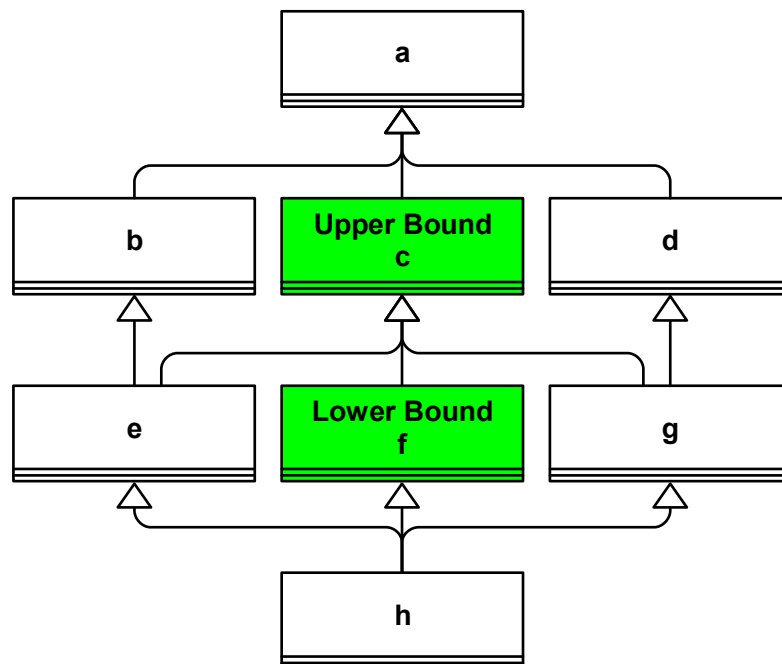
- Eine Typvariable τ steht für eine bestimmte **Menge von Referenztypen**, die durch untere und obere Schranken definiert wird
- Eine Typvariable τ kann selbst als **Referenztyp** verwendet werden, d.h. Variablen bzw. Ausdrücke können vom Typ τ sein
- Zuweisungen unter Beteiligung von Variablen bzw. Ausdrücken des Typs τ sind eingeschränkt und müssen für alle Typen der Typmenge τ sicher sein

```
 $\tau$  right = ...;  
v left = right;  
v :> u, u  $\in$  UB( $\tau$ )
```

Existieren **mehrere** obere Schranken UB(τ), so entspricht die Zuweisung **von** einem Ausdruck des Typs τ einer Zuweisung von einer beliebigen **oberen Schranke** $u \in$ UB(τ)



- Eine Typvariable \mathbf{T} steht für eine bestimmte **Menge von Referenztypen**, die durch untere und obere Schranken definiert wird
- Eine Typvariable \mathbf{T} kann selbst als **Referenztyp** verwendet werden, d.h. Variablen bzw. Ausdrücke können vom Typ \mathbf{T} sein
- Zuweisungen unter Beteiligung von Variablen bzw. Ausdrücken des Typs \mathbf{T} sind eingeschränkt und müssen für alle Typen der Typmenge \mathbf{T} sicher sein



Außerdem sind Zuweisungen von Ausdrücken des Typs \mathbf{T} auf Variablen des Typs \mathbf{T} möglich, wenn \mathbf{T} jeweils nachweislich denselben tatsächlichen Typen bezeichnet

```
 $\mathbf{T}$  right = ...;  
 $\mathbf{T}$  left  = right;
```

- Zwei Anwendungen für Typvariablen bei Java Generics
 - **Typparameter** (generische Klassen / Schnittstellen / Methoden)
 - **Parametrisierte Typen** mit Wildcards
- **Beispiel:** Typparameter **E** in der Klasse **SomeClass**
 - **E** ist eine **Typvariable**, Zuweisungen unter Beteiligung des Typs **E** unterliegen den Beschränkungen der Typschränken (Null-Typ und **object**)
 - Signaturen von **push** und **pull** **innerhalb** von **SomeClass**:

```
public int push(E elem)
public E    pull()
```
 - **this.push(this.pull())** zulässig, da **E** hier nachweislich denselben tatsächlichen Typen bezeichnet

```
public class SomeClass<E> {
    public int push(E elem) { /* ... */ }

    public E pull() { /* ... */ }

    public int strange() {
        return this.push( this.pull() );
    }
}
```


- Zwei Anwendungen für Typvariablen bei Java Generics
 - **Typparameter** (generische Klassen / Schnittstellen / Methoden)
 - **Parametrisierte Typen** mit Wildcards

- **Beispiel:** Typparameter **E**
in der Klasse **SomeClass**

- Wird **someClass** im Rahmen einer Variablendeklaration mit einem **konkreten Typen** parametrisiert, so ersetzt dieser konkrete Typ in den Signaturen die Typvariable **E**

- Signaturen von **push** und **pull** für **SomeClass<Integer>**:
`public Integer push(Integer elem)`
`public Integer pull()`

- `sca.push(sca.pull())` offensichtlich zulässig

```
public class SomeClass<E> {  
    public int push(E elem) { /* ... */ }  
  
    public E pull() { /* ... */ }  
  
    public int strange() {  
        return this.push( this.pull() );  
    }  
}
```

```
public static void main(String[] args) {  
    SomeClass<Integer> sca = ...;  
    sca.push( sca.pull() );  
}
```

- Zwei Anwendungen für Typvariablen bei Java Generics
 - **Typparameter** (generische Klassen / Schnittstellen / Methoden)
 - **Parametrisierte Typen** mit Wildcards
- **Beispiel:** Typparameter **E** in der Klasse **SomeClass**
 - Wird **someClass** im Rahmen einer Variablendeklaration mit einer **Wildcard** parametrisiert, so ersetzt eine fiktive Typvariable α in den Signaturen die Typvariable **E**
 - Signaturen von **push** und **pull** für **SomeClass<? extends Number>**:
`public α push(α elem)`
`public α pull()`
 - Obere Schranke von α ist **Number**, untere Schranke ist der Null-Typ

```
public class SomeClass<E> {  
    public int push(E elem) { /* ... */ }  
  
    public E pull() { /* ... */ }  
  
    public int strange() {  
        return this.push( this.pull() );  
    }  
}
```

```
public static void main(String[] args) {  
    SomeClass<? extends Number> scb = ...;  
  
    Number n = scb.pull();  
    scb.push(null);  
}
```

- Zwei Anwendungen für Typvariablen bei Java Generics
 - **Typparameter** (generische Klassen / Schnittstellen / Methoden)
 - **Parametrisierte Typen** mit Wildcards

- **Beispiel:** Typparameter **E**
in der Klasse **SomeClass**

- Wird **someClass** im Rahmen einer Variablendeklaration mit einer **Wildcard** parametrisiert, so ersetzt eine fiktive Typvariable α in den Signaturen die Typvariable **E**

```
public class SomeClass<E> {  
    public int push(E elem) { /* ... */ }  
  
    public E pull() { /* ... */ }  
  
    public int strange() {  
        return this.push( this.pull() );  
    }  
}
```

- Signaturen von **push** und **pull**
für **SomeClass<? extends Number>**:
public α push(α elem)
public α pull()

```
public static void main(String[] args) {  
    SomeClass<? extends Number> scb = ...;  
    SomeClass<? extends Number> scc = ...;  
    scc.push( scb.pull() ); /* Fehler */  
}
```

- Für jede Verwendung entsteht eine **eigenständige** fiktive Typvariable, d.h. die Typvariablen für **scc.push(...)** und **scb.pull()** sind nicht identisch

- Zwei Anwendungen für Typvariablen bei Java Generics
 - **Typparameter** (generische Klassen / Schnittstellen / Methoden)
 - **Parametrisierte Typen** mit Wildcards

- **Beispiel:** Typparameter **E**
in der Klasse **SomeClass**

- Wird **someClass** im Rahmen einer Variablendeklaration mit einer **Wildcard** parametrisiert, so ersetzt eine fiktive Typvariable α in den Signaturen die Typvariable **E**
- Signaturen von **push** und **pull** für **SomeClass<? extends Number>**:
`public α push(α elem)`
`public α pull()`
- Auch für Aufrufe auf derselben Variablen entstehen eigenständige Typvariablen, d.h. die Typvariablen für **scc.push(...)** und **scc.pull()** sind nicht identisch

```
public class SomeClass<E> {  
    public int push(E elem) { /* ... */ }  
  
    public E pull() { /* ... */ }  
  
    public int strange() {  
        return this.push( this.pull() );  
    }  
}
```

```
public static void main(String[] args) {  
    SomeClass<? extends Number> scb = ...;  
    SomeClass<? extends Number> scc = ...;  
    scc.push( scc.pull() ); /* Fehler */  
}
```


↳ Da sich d. Instanz theoretisch verändern könnte (neuer Typ α)

1. **Etwas Wiederholung zu Java...**
2. **Fortgeschrittene Generics**
3. **Verschachtelte Typen**
4. **Funktionale Interfaces**
5. **Funktionale Programmierung mit Java Streams**
6. **Nebenläufigkeit**
7. **Reflection & Annotations**

■ Idee hinter Iteratoren

- **Traversierung von Containern** generalisieren, d.h. einheitliche systematische Traversierung der Elemente unabhängig vom Containertyp
- **Mehrere unabhängige Traversierungen** zu einem Container gleichzeitig, d.h. 1 Containerinstanz => n Traversierungsinstanzen

■ **Iterator** (Stand Java 7):

- **Iterator<E>**: Schnittstelle zur sequenziellen Verarbeitung der Elemente einer **Collection**
- **hasNext()**: Liefert **true** solange unverarbeitete Collection-Elemente existieren
- **next()**: Liefert das jeweils nächste Element der Collection und setzt den Cursor weiter; wirft eine **NoSuchElementException**, wenn in der Collection kein unverarbeitetes Element existiert (**!hasNext()**)
- **remove()**: Entfernt das **zuletzt** per **next()** ausgelesene Element der Collection
 - **UnsupportedOperationException** - Der Iterator unterstützt die Operation nicht 
 - **IllegalStateException** - **next()** wurde nach dem letzten **remove()** noch nicht wieder aufgerufen (oder noch gar nicht)

```
public interface Iterator<E> {  
    boolean hasNext();  
    E        next();  
    void     remove();  
}
```

- Definiertes Interface für die standardisierte **Anforderung** einer **Iterator**-Instanz:
Iterable (Stand Java 7)

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

- **Beispiel**

- Es gelte `List<E>` implements `Iterable<E>`, d.h. die Listenklassen implementieren die Methode `Iterator<E> iterator()`
- Implementierung von `sum()` per Iterator

```
public static int sum(List<? extends Number> l) {  
    int result = 0;  
    Iterator<? extends Number> it = l.iterator();  
    while(it.hasNext()) {  
        result += it.next().intValue();  
    }  
    return result;  
}
```

Beispiel: Iterator für eigenen Listentypen (1)

```
public interface MyList<E> {
    MyList<E> insert(E value);
    MyList<E> remove(int idx);
    E get(int idx);
    Integer indexOf(E value, int start_idx);
    boolean contains(E value);
    int count(E value);
    int length();
    boolean isEmpty();
}
```

```
public class MyLinkedList<E>
implements MyList<E> {

    private static class ListNode<V> {
        public V value;
        public ListNode<V> next = null;

        public ListNode(V value, ListNode<V> next) {
            this.value = value;
            this.next = next;
        }
    }

    private ListNode<E> first = null;

    /* ... Operationsimplementierungen ... */

}
```


▪ Erste Implementierung einer Iterator-Klasse zu unserer `MyList<E>`:

- Verwaltung des aktuellen Listen-Index
- Aufruf der List-Methoden (`length()`, `get(int)`)
- **Kein Zugriff auf Listen-Implementierung möglich**
- **Aber:** Für jeden Untertyp von `MyList<E>` verwendbar
- **Achtung:** Hier und nachfolgend als Vereinfachung keine Berücksichtigung der `NoSuchElementException`

```
public class ListIterator<E> implements Iterator<E> {  
  
    private int curr_idx;  
    private MyList<E> l;  
  
    public ListIterator(MyList<E> l) {  
        this.l = l;  
    }  
  
    @Override  
    public boolean hasNext() {  
        return this.curr_idx < this.l.length();  
    }  
  
    @Override  
    public E next() {  
        E result = this.l.get(this.curr_idx);  
        this.curr_idx++;  
        return result;  
    }  
}
```

- Erzeugung des Iterators auf Ebene von `MyList<E>` mittels Default-Methode:

```
public interface MyList<E> extends Iterable<E> {  
  
    /* ... */  
  
    @Override  
    default Iterator<E> iterator() {  
        return new ListIterator<E>(this); /* Übergabe des Listenobjektes per this */  
    }  
  
}
```

- Anwendungsbeispiel:

```
MyList<Pair<?>> lp = new MyLinkedList<>();  
/* ... */  
for(Pair<?> p : lp) {  
    System.out.println(p);  
}
```

- Besonderheit des bisherigen `ListIterator<E>`: Eigenständige Klasse losgelöst von `MyList<E>` bzw. entsprechenden Implementierungen
- **Kein Zugriff auf geschützte Datenfelder** (oder Methoden), dadurch kein Zugriff auf die jeweilige Listenimplementierung (z.B. Verkettungsstruktur)

=> **ineffiziente Umsetzung** per indiziertem Zugriff, Traversierungszustand geht nach jedem Element wieder verloren (wohlgemerkt implementationsspezifisches Problem, für eine `MyArrayList` stellte dies keinen Nachteil dar)
- Gefahr der **separaten Instanziierung**: Die Methode `iterator()` aus `Iterable<E>` soll als **Fabrikmethode** fungieren, die Auswahl der Implementierung also gekapselt werden. Ein Austausch der `iterator`-Implementierung würde für Client-Code insofern transparent erfolgen. **Im Beispiel aber könnte Clientcode den Iterator selbst instanziierten, die Wartbarkeit des Systems leidet.**
- **Ziel**: Die `iterator()`-Methode sollte `Iterator<E>`-Instanzen liefern, die auf die jeweils genutzte Implementierung (z.B. `MyLinkedList<E>`) zurückgreifen

- **Lösungsmöglichkeit:** Deklaration von `Iterator<E>`-Klassen jeweils als **Nested Class** in den Implementierungsklassen (z.B. `MyLinkedList<E>`)
- **Nested Type (Nested Class / Nested Interface):** Deklaration eines Typen im Rumpf einer **äußeren** Klasse oder eines **äußeren** Interfaces

Ansonsten: **Top Level Type** (Deklaration typischerweise im Rahmen einer eigenen Java-Quelldatei)

- Differenzierung von **Nested Classes**
 - **Member Classes:** Deklaration **direkt** innerhalb einer Klasse / eines Interfaces
 - **Local Class:** Deklaration einer Klasse **in einem Codeblock**
 - **Anonymous Class:** Implizite Deklaration einer Klasse **bei Instanziierung**
- **Nested Interfaces** sind immer **Member Interfaces**, d.h. keine Interface-Deklarationen in Codeblöcken oder im Rahmen der Instanziierung

- Differenzierung von **Member Classes**
 - **Non-Inner Member Class:** Deklaration **mit** Schlüsselwort `static`, kein Zugriff auf **Instanzvariablen** einer äußeren Klasse
 - **Inner Member Class:** Deklaration **ohne** Schlüsselwort `static`, Zugriff auf **Instanzvariablen** einer äußeren Klasse
- **Member Interfaces** sind immer **Non-Inner** Member Interfaces (das Schlüsselwort `static` darf dabei redundant angegeben werden), d.h. es gibt keine "inneren" Interfaces
- **Non-Inner Member Class**
 - Abgesehen vom Deklarationsort ähnlich zu Top-Level-Klasse
 - **Aber:** Kann per Schutzstufe vor dem Zugriff anderer Typen geschützt werden
 - **Aber:** Kann auf geschützte Member (Methoden, Instanzvariablen, ...) einer äußeren Klasse zugreifen

- `LinkedListIt<E>` als Non-Inner Member Class von `MyLinkedList<E>`

```
public class MyLinkedList<E> implements MyList<E> { Gängige Praxis: immer unterschiedliche Bezeichner verwenden.
    private static class LinkedListIt<V> implements Iterator<V> {

        private ListNode<V> cursor = null; /* Zugriff auf priv. Member ListNode */

        private LinkedListIt(MyLinkedList<V> l) {
            this.cursor = l.first;          /* Zugriff auf priv. Member first */
        }

        @Override
        public boolean hasNext() {
            return this.cursor != null;
        }

        @Override
        public V next() {
            V result = this.cursor.value;
            this.cursor = this.cursor.next;
            return result;
        }
    }
}

/* ... */
```

- `LinkedListIt<E>` als Non-Inner Member Class von `MyLinkedList<E>` (Forts.)

```
public abstract class MyLinkedList<E> implements MyList<E> {  
  
    /* ... */  
  
    @Override  
    public Iterator<E> iterator() {  
        return new LinkedListIt<E>(this);  
    }  
  
}
```

▪ Eigenschaften

- Geschützte Deklaration im Rumpf von `MyLinkedList<E>`, kein Zugriff (Instanziierung, Beerben, ...) außerhalb von `MyLinkedList<E>` möglich
- Zugriff auf geschützte Member (hier: `ListNode`-Klasse und `first`-Variable)
- **Aber:** `LinkedListIt<E>`-Klasse nur für `MyLinkedList<E>` verwendbar

▪ Inner Member Class (innere Klasse)

- Eine Instanz der inneren Klasse wird ausgehend **von einer Instanz der äußeren Klasse** erzeugt, aus der inneren Instanz besteht Zugriff auf die äußere Instanz
- Zugriff auf äußere Instanz per `<ÄußereKlasse>.this`

```
public class Out {  
    public class In {  
        private int b;  
  
        public In(int b) {  
            this.b = b;  
        }  
  
        public int getB() {  
            return this.b;  
        }  
  
        public int getA() {  
            return Out.this.a;  
        }  
    }  
    /* ... */  
}
```

```
/* ... */  
private int a;  
  
public Out(int a) {  
    this.a = a;  
}  
  
public In getInner(int b) {  
    return new In(b);  
}  
}
```

Instanziierung **implizit** ausgehend von der aktuellen Instanz von Out (**this**)

■ Inner Member Class (innere Klasse) (Forts.)

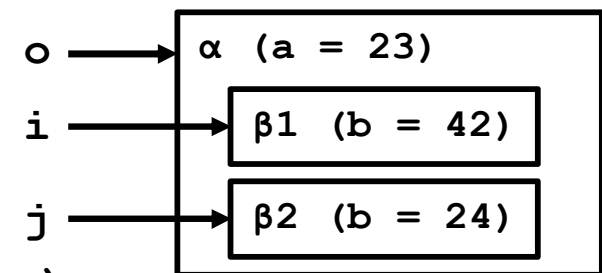
- Beispiel mit `out` / `in` (Codefragment außerhalb von `Out`):

```
Out      o = new Out(23);      /* erzeugt eine Out-Instanz  $\alpha$  */
Out.In   i = o.getInner(42);   /* erzeugt eine In-Instanz  $\beta_1$  zu  $\alpha$  */
System.out.println(i.getA()); /* Ausgabe der Variable a von  $\alpha$  */
> 23
System.out.println(i.getB()); /* Ausgabe der Variable b von  $\beta_1$  */
> 42

Out.In   j = new Out.In(24); /* Compilerfehler: Keine äußere Instanz */
Out.In   j = o.new In(24);     /* erzeugt eine 2.In-Instanz  $\beta_2$  zu  $\alpha$  */
System.out.println(j.getA()); /* Ausgabe der Variable a von  $\alpha$  */
> 23
System.out.println(j.getB()); /* Ausgabe der Variable b von  $\beta_2$  */
> 24
```

- Erzeugen einer Instanz einer inneren Klasse:

- `x.new A(...)`, wobei `x` eine Referenz der äußeren und `A` ein Konstruktor der inneren Klasse ist (z.B. `o.new In(24)`)
- Ohne Referenz wird implizit `this` verwendet, wenn `this` vom Typ der äußeren Klasse ist: `new A(...) => this.new A(...)`



■ Praktischer Anwendungsfall:

```
public class Person {  
    public class Address {  
        private String street;  
  
        /* ... */  
        public String toString() {  
            return Person.this.name + ", Adresse: " + this.street;  
        }  
    }  
  
    private String name;  
    private Address[] addresses;  
  
    /* ... */  
    public Address getAddress(...) { /* ... */ }  
    public Address createAddress(...) { /* ... */ }  
}
```

```
Person          p = new Person("Musterperson");  
Person.Address a = p.createAddress("Some Street");  
System.out.println(a);  
> Musterperson, Adresse: Musterweg
```

- `LinkedListIt<E>` als Inner Class von `MyLinkedList<E>`

```
public abstract class MyLinkedList<E> implements MyList<E> {  
    private class LinkedListIt implements Iterator<E> {  
  
        private ListNode<E> cursor = null;  
  
        private LinkedListIt() {  
            this.cursor = MyLinkedList.this.first;  
        }  
  
        @Override  
        public boolean hasNext() {  
            return this.cursor != null;  
        }  
  
        @Override  
        public E next() {  
            E result = this.cursor.value;  
            this.cursor = this.cursor.next;  
            return result;  
        }  
    }  
}  
/* ... */
```

Ohne static:
Deklaration einer
inneren Klasse

Typparameter fehlt:
Als innere Klasse kann die
Typvariable E der
äußeren Klasse genutzt
werden

Äußere Instanz:
Zugriff auf die
MyLinkedList<E>-
Instanz, in die das
LinkedListIt-Objekt
eingebettet ist

- `LinkedListIt<E>` als Inner Class von `MyLinkedList<E>` (Forts.)

```
public abstract class MyLinkedList<E> implements MyList<E> {  
    private class LinkedListIt implements Iterator<E> {  
  
        private ListNode<E> cursor = MyLinkedList.this.first;  
  
        @Override  
        public boolean hasNext() {  
            return this.cursor != null;  
        }  
  
        @Override  
        public E next() {  
            E result = this.cursor.value;  
            this.cursor = this.cursor.next;  
            return result;  
        }  
    }  
}  
/* ... */
```

Kein Konstruktor mehr
erforderlich:
Die Listeninstanz wird nicht
als Parameter übergeben,
sondern steht bereits
**implizit als äußere
Instanz** zur Verfügung

- `LinkedListIt<E>` als Inner Class von `MyLinkedList<E>` (Forts.)

```
public abstract class MyLinkedList<E> implements MyList<E> {  
  
    /* ... */  
  
    @Override  
    public Iterator<E> iterator() {  
        return new LinkedListIt();  
        /* entspricht this.new LinkedListIt() */  
    }  
  
}
```

Kein Parameter mehr erforderlich, `this`-Referenz der `iterator()`-Methode wird automatisch zur äußeren Instanz

- Im Rahmen der Instanziierung wird automatisch die aktuelle Listeninstanz zur äußeren Instanz des neuen `LinkedListIt`-Objektes

=> dadurch auch keine Angabe von Typargumenten beim Konstruktoraufruf mehr notwendig
- Im Wesentlichen **Notations-Vereinfachung**: Implizite Verwaltung der äußeren Instanz statt Parameter-Übergabe

▪ Local Class

- Deklaration nicht direkt als **Member** eines anderen Typen, sondern innerhalb eines **Codeblocks** (zur Klarstellung: dennoch **Nested Class**, da der umgebende Codeblock und damit mittelbar auch die **Local Class** im Rumpf einer anderen Typdeklaration stehen müssen)
- Wie **Inner Class**, aber:
 - Auch in statischem Kontext möglich, in diesem Fall wird keine äußere Instanz vorgesehen
 - **Sichtbarkeitsbereich** auf den deklarierenden Codeblock **eingeschränkt**
 - **Zugriff auf lokale final Variablen** des Codeblocks
hier: Methoden-Parameter **b**

```
public class Out {  
    private int a;  
    public Object strange(int b) {  
  
        class Local {  
            public String toString() {  
                return Integer.toString(  
                    Out.this.a + b);  
            }  
        }  
  
        return new Local();  
    }  
}
```

```
Out o = new Out(23);  
System.out.println( o.strange(123) );  
> 146
```

▪ Local Class (Forts.)

- Parameter **b** ist doch nicht `final`?
- Für Local Classes genügt es, wenn Variablen "**effectively final**" sind
- **1. Fall:** Variable **v** mit Initializer (z.B. auch Methodenparameter)
 - Keine Zuweisung mit der Variablen **v** auf der linken Seite
 - **v** ist kein Operand eines Prefix oder Postfix Inkrement / Dekrement Operators (`++` / `--`)
- **2. Fall:** Variable **v** ohne Initializer
 - Zuweisungen mit der Variablen **v** auf der linken Seite nur an Stellen, an denen **v** "definitely unassigned" ist
 - **v** ist kein Operand eines Prefix oder Postfix Inkrement / Dekrement Operators (`++` / `--`)

```
public class Out {  
    private int a;  
    public Object strange(int b) {  
  
        class Local {  
            public String toString() {  
                return Integer.toString(  
                    Out.this.a + b);  
            }  
        }  
  
        return new Local();  
    }  
}
```

- `LinkedListIt<E>` als Local Class von `MyLinkedList<E>.iterator()`

```
public abstract class MyLinkedList<E> implements MyList<E> {
    /* ... */
    public Iterator<E> iterator() {
        class LinkedListIt implements Iterator<E> {
            private ListNode<E> cursor = MyLinkedList.this.first;

            @Override
            public boolean hasNext() {
                return this.cursor != null;
            }

            @Override
            public E next() {
                E result = this.cursor.value;
                this.cursor = this.cursor.next;
                return result;
            }
        }
        return new LinkedListIt();
    }
}
```


▪ Anonymous Class

- Wie Local Class, aber **implizite Klassendeklaration** ohne Namen (daher **anonym**) zur Verwendung an einer einzigen Stelle im Programmcode
- **Anwendungsfall:** Implementierung eines Interfaces bzw. Spezialisierung einer Klasse **ohne Wiederverwendungswert** außerhalb der Verwendungsstelle
- **Beispiel:** `Transformer / Pair<E>`
 - Quadrieren von `Integer`-Werten eines Paares
 - Einmalige Implementierung eines entsprechenden Transformers im Rahmen des Aufrufs von `Pair.apply()`

```
public interface Transformer<S,T> {  
    T transform(S input);  
}
```

```
public class Pair<E> {  
    /* ... */  
    public <F> Pair<F> apply(Transformer<? super E, ? extends F> t) { /* ... */ }  
}
```

▪ Anonymous Class (Forts.)

- Angabe des zu implementierenden Interfaces bzw. der zu spezialisierenden Klasse im Rahmen eines Konstruktoraufrufs

hier: `new Transformer<Integer,Integer>()`

- Angabe des Klassenrumpfes unmittelbar **nach dem Konstruktoraufruf**

```
Pair<Integer> pi = new Pair<Integer>(23, 42);  
pi = pi.apply( new Transformer<Integer,Integer>() {  
    @Override  
    public Integer transform(Integer input) {  
        return input * input;  
    }  
} );  
System.out.println(pi);  
> 529:1764
```

- Klassenrumpf enthält im Beispiel die gewünschte Implementierung von `transform()`
- Implementierung erhält keinen Klassennamen und kann an anderer Stelle nicht instanziiert oder als Typ verwendet werden

- Anonyme Klasse in `LinkedList<E>.iterator()`

```
public abstract class MyLinkedList<E> implements MyList<E> {
    /* ... */
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private ListNode<E> cursor = MyLinkedList.this.first;

            @Override
            public boolean hasNext() {
                return this.cursor != null;
            }

            @Override
            public E next() {
                E result = this.cursor.value;
                this.cursor = this.cursor.next;
                return result;
            }
        };
    }
}
```

Arten von geschachtelten Typen mit jeweiligen Besonderheiten

Nested Type	Deklarationsort			
	<i>Interface</i>	<i>Class</i>	<i>Codeblock</i>	<i>Instanziierung</i>
Interface	Non-Inner Member Interface (implizit <code>public</code> und <code>static</code> , Modifier-Angabe erlaubt)	Non-Inner Member Interface (implizit <code>static</code> , Schutzstufe <code>public</code> , <code>protected</code> oder <code>private</code>)		
Class	Non-Inner Member Class (implizit <code>public</code> und <code>static</code> , Modifier-Angabe erlaubt)	Non-Inner Member Class (Schlüsselwort <code>static</code>)	Local Class (immer Inner Class, keine Angabe von <code>static</code> oder Schutzstufen erlaubt)	Anonymous Class (immer Inner Class)
		Inner Member Class		