

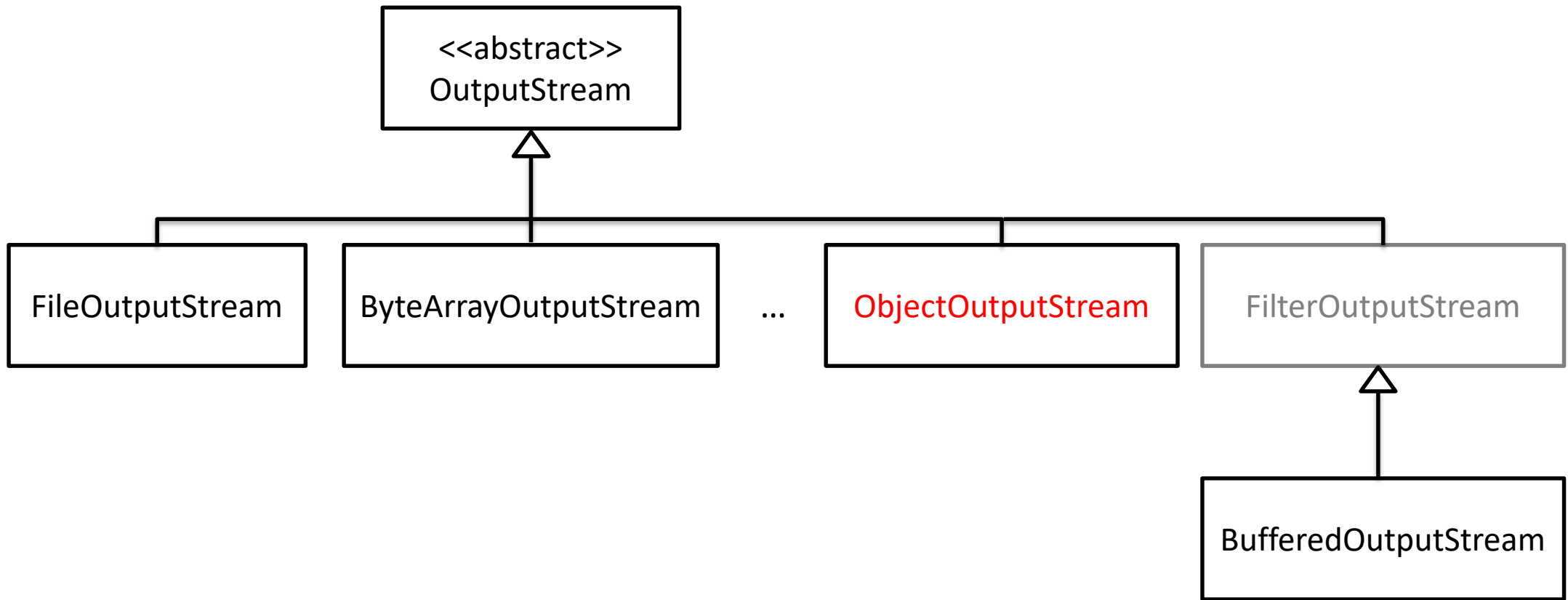
Lektion 19

Byte Streams: ObjectOutputStream ObjectInputStream
Generics

Byte Streams:

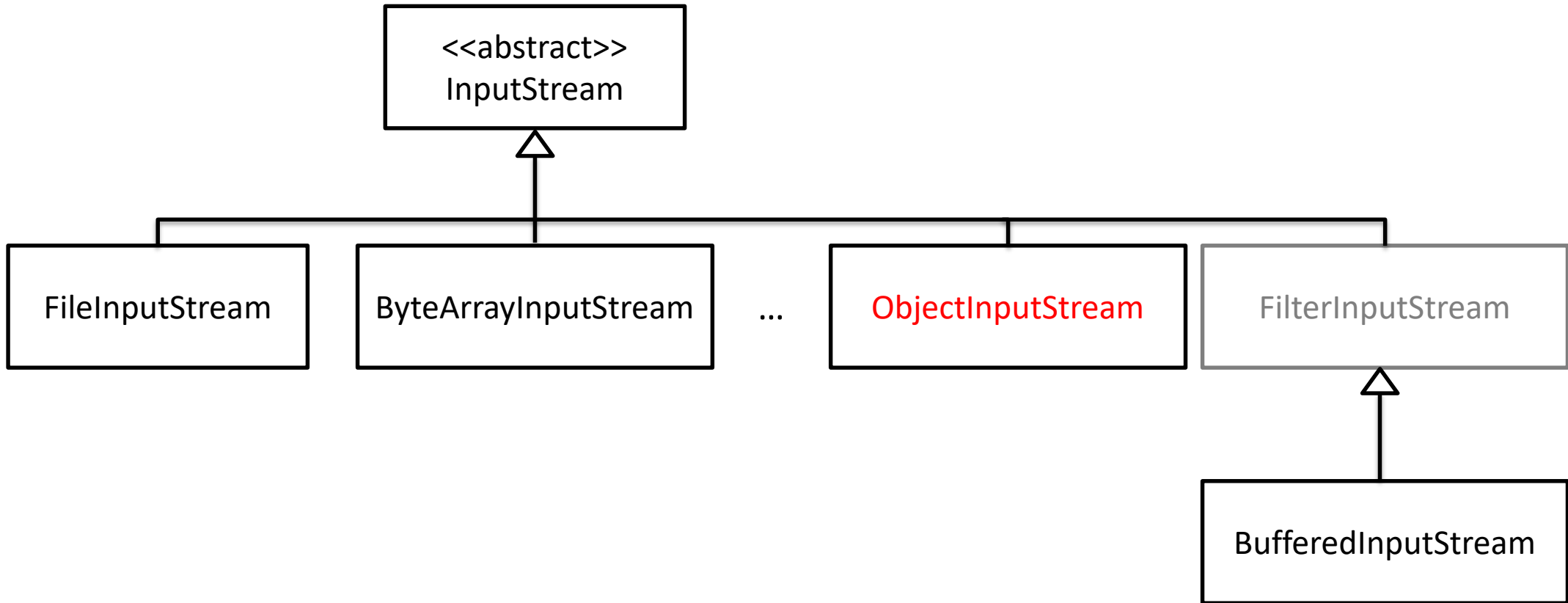
ObjectOutputStream

ObjectInputStream



Ein `ObjectOutputStream` überführt den Zustand eines Objekts in eine Abfolge von Bytes.

Dieser Vorgang heißt **Serialisierung**.



Ein `ObjectInputStream` stellt den Zustand eines Objekts aus einer Abfolge von Bytes wieder her.

Dieser Vorgang heißt **Deserialisierung**.

Serialisierbarkeitsbedingungen

- Ein Objekt ist genau dann serialisierbar, wenn dessen Klasse folgende Bedingungen erfüllt:
 - Die Klasse implementiert das Interface `java.io.Serializable`
 - Die Klasse muss auf den Standardkonstruktor der ersten nichtserialisierbaren Oberklasse zugreifen können.
 - Die zu serialisierenden Attribute müssen serialisierbar sein.
- Die Klasse kann nicht zu serialisierende (bzw. zu serialisierende) Attribute kennzeichnen.

Das Interface `Serializable` ist leer und dient nur zur Markierung.

```
public interface Serializable
{
}
```

Warum sind also nicht alle Klassen serialisierbar?

- Objekte wie Threads enthalten plattformabhängige Informationen
- Sicherheitsrelevante Daten (z.B. Passwörter) sollten nicht unverschlüsselt über das Netz übertragen oder auf Festplatte gespeichert werden

Ein Objekt in einer Datei speichern

```
public class Person implements Serializable
{
    String name;
    String vorname;

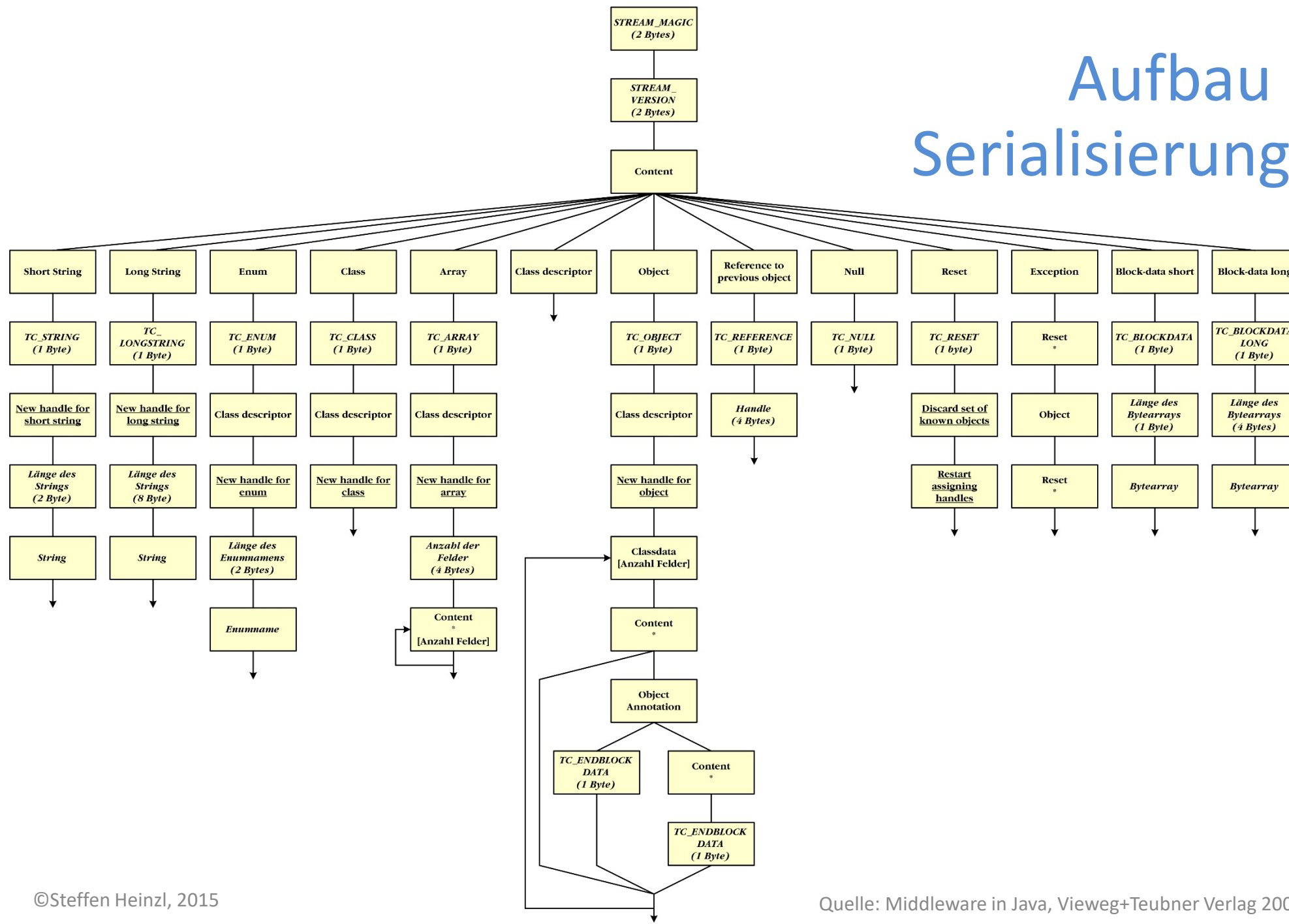
    public Person(String name, String vorname)
    {
        this.name = name;
        this.vorname = vorname;
    }
}
```

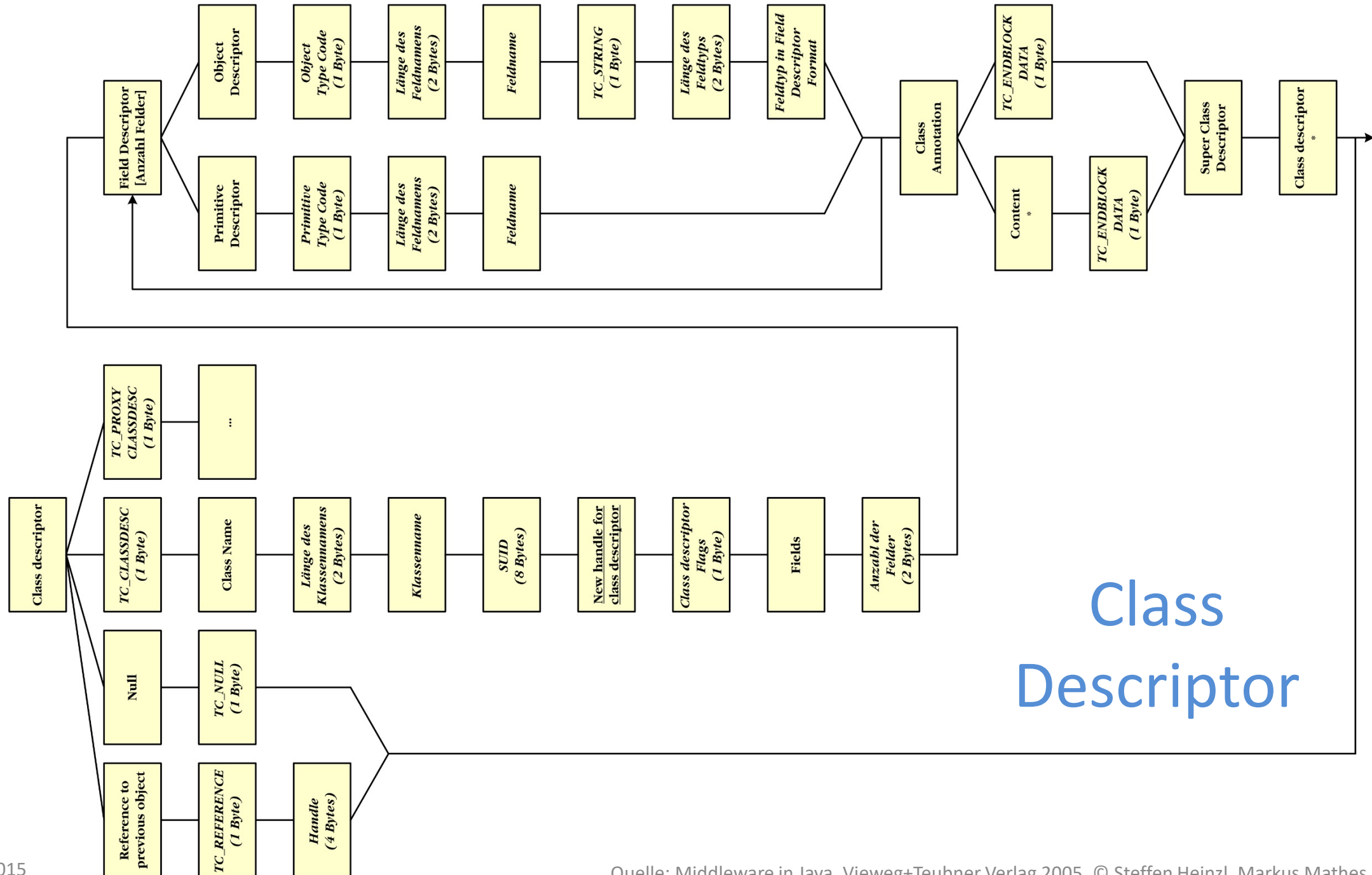
```
public class WritePerson
{
    public static void main(String[] args)
    {
        try
        {
            Person p = new Person("Doe", "John");
            ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("person.dat"));
            oos.writeObject(p);
            oos.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

 person.dat ✕

```
1-í  sr streams.objectstreams.Person^û\Ù`1fM  L  namet  Ljava/lang/String;L  vor
```


Aufbau des Serialisierungsstreams





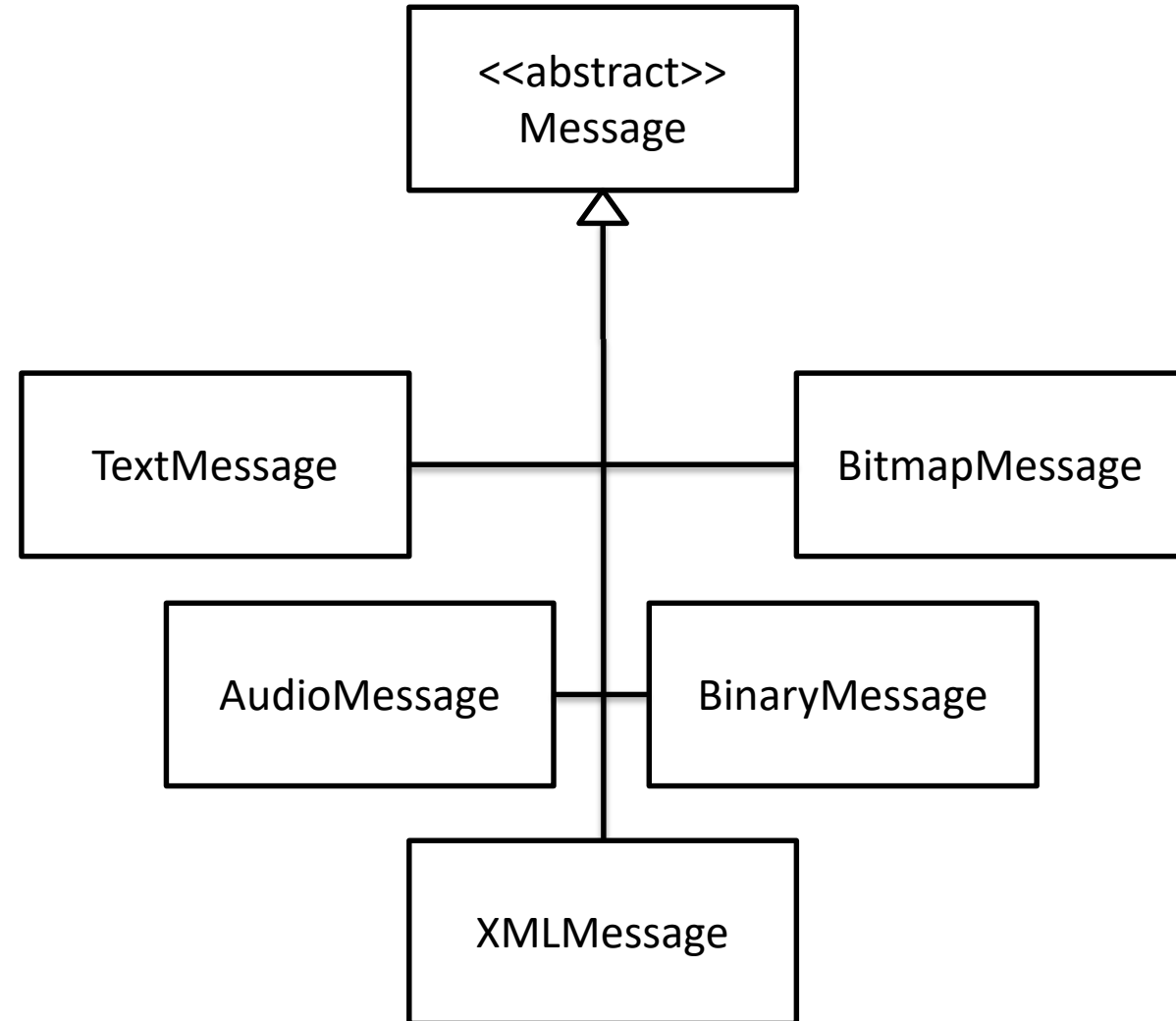
Ein (geschriebenes) Objekt aus einer Datei lesen

```
public class ReadPerson
{
    public static void main(String[] args)
    {
        try
        {
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.dat"));
            Person person = (Person) ois.readObject();
            ois.close();
            System.out.println(person.vorname + " " + person.name);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

readObject wirft eine EOFException, wenn der letzte Datensatz gelesen wurde (analog zur -1 der read()-Methode)

Beispiel für Nachrichtenmodell zum Austausch von Nachrichtenobjekten

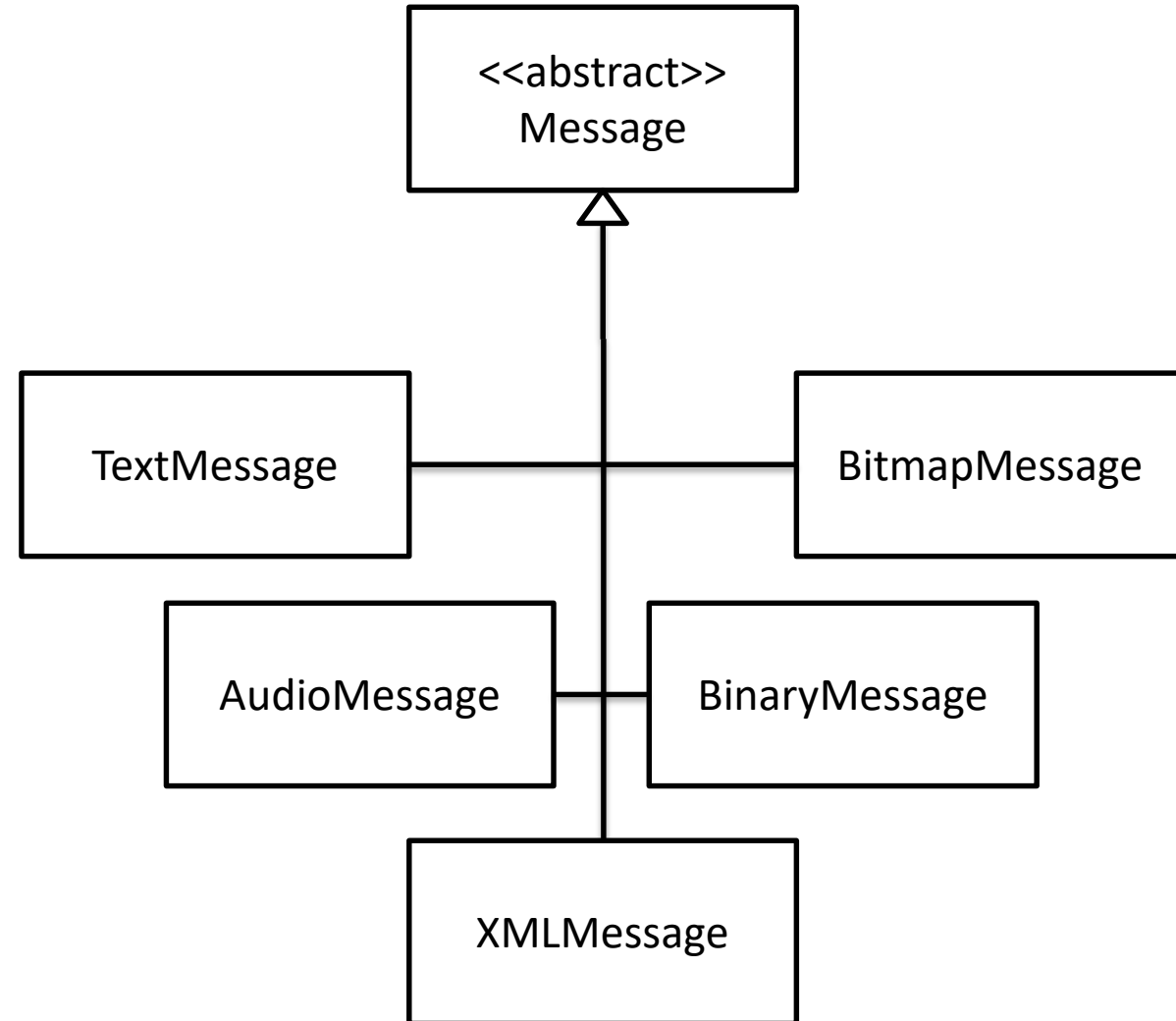
- Oberklasse Message
- Unterklassen für verschiedene Datentypen



Beispiel für Nachrichtenmodell zum Austausch von Nachrichtenobjekten

- Überprüfung anhand der Klasse, welches Objekt gesendet wurde

```
...  
Message msg = (Message) ois.readObject();  
if (msg.getClass() == TextMessage.class)  
{  
    TextMessage tm = (TextMessage) msg;  
    ...  
}
```

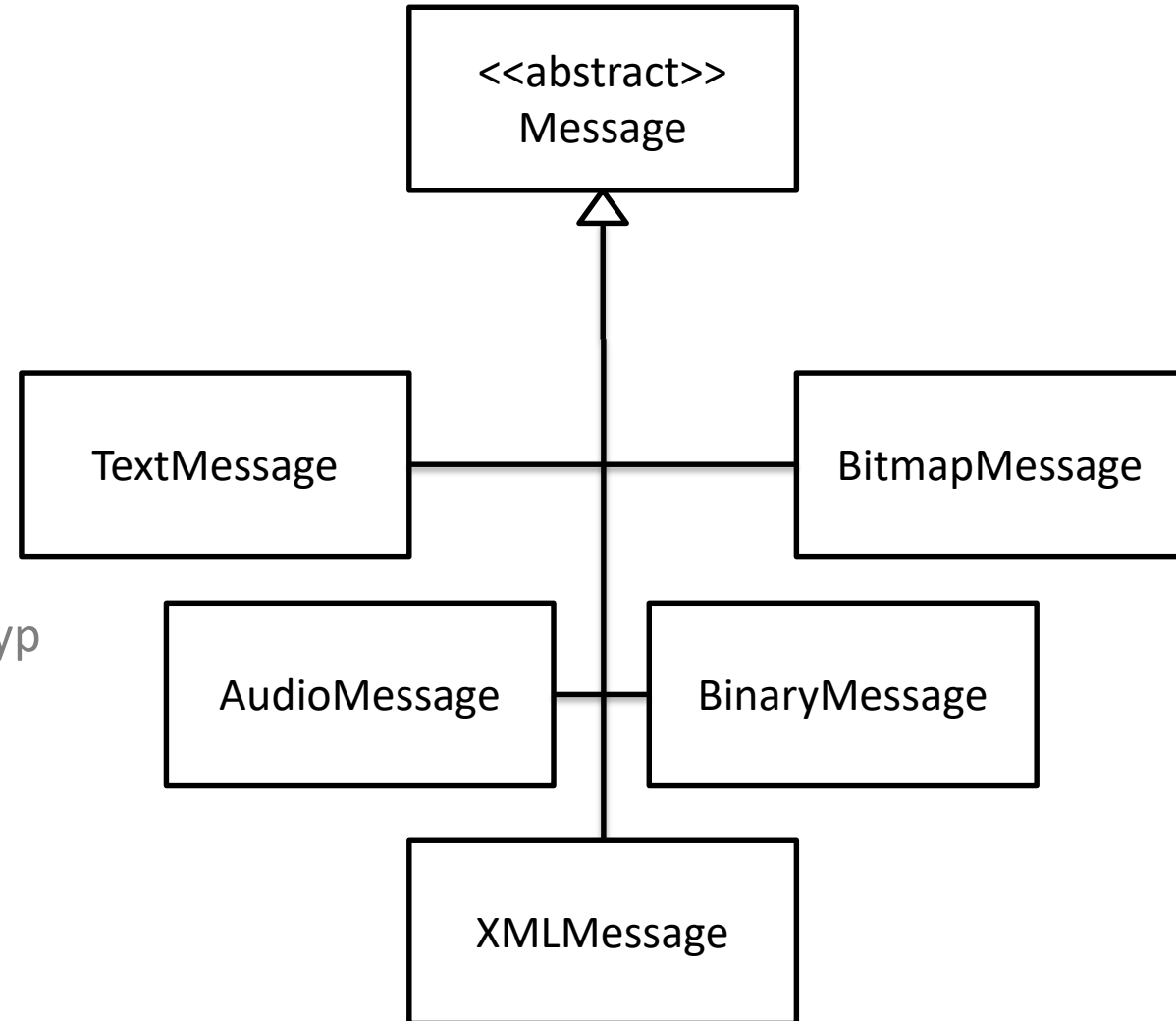


Beispiel für Nachrichtenmodell zum Austausch von Nachrichtenobjekten

- Überprüfung anhand der Vererbungshierarchie, welches Objekt gesendet wurde

```
...  
Message msg = (Message) ois.readObject();  
if (msg instanceof TextMessage)  
{  
    TextMessage tm = (TextMessage) msg;  
    ...  
}
```

- **instanceof** liefert true, wenn msg ein Objekt vom Typ TextMessage oder von einer Unterklasse von TextMessage ist.



Standardmäßig werden alle Attribute serialisiert.

Falls ein Attribut nicht serialisierbar ist, wird eine *NotSerializableException* ausgelöst.


Das Schlüsselwort **transient** (flüchtig) schließt Attribute von der Serialisierung aus.

```
import java.io.Serializable;
public class MySerializableObject implements Serializable
{
    int integer;
    transient Thread thread;
}
```

```
import java.io.Serializable;

public class Person implements Serializable
{
    String name;
    String vorname;

    public Person(String name, String vorname)
    {
        this.name = name;
        this.vorname = vorname;
    }
}
```



Eine serialisierbare Klasse erzeugt ein Warning, da die serialVersionUID fehlt.

```
import java.io.Serializable;

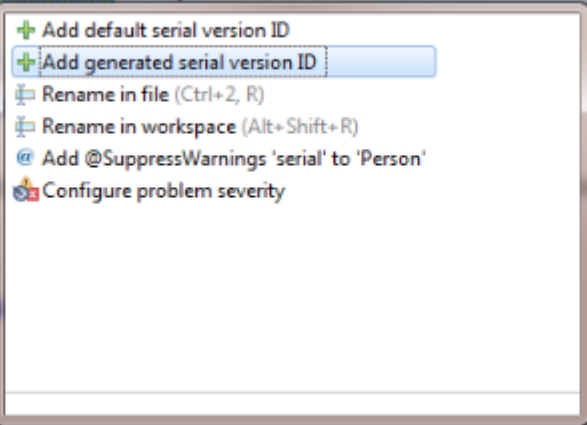
The serializable class Person does not declare a static final serialVersionUID field of type long
public class Person implements Serializable
{
    String name;
    String vorname;

    public Person(String name, String vorname)
    {
        this.name = name;
        this.vorname = vorname;
    }
}
```



```
public class Person implements Serializable
```

```
{  
    String name;  
    String vorname;  
  
    public Person(  
    {  
        this.name = name;  
        this.vorname = vorname;  
    }  
}
```



Adds a generated serial version ID to the selected type.

Use this option to add a compiler-generated ID if the type did not undergo structural changes since its first release.

Press 'Tab' from proposal table or click for focus



```
public class Person implements Serializable
```

```
{  
    private static final long serialVersionUID = -8576158976705468851L;  
  
    String name;  
    String vorname;  
  
    public Person(String name, String vorname)  
    {  
        this.name = name;  
        this.vorname = vorname;  
    }  
}
```

serialVersionUID wird bei der Serialisierung von einem Objekt einer Klasse berechnet.

Um die Serialisierung zu beschleunigen, sollte man die serialVersionUID generieren lassen, sobald die Klasse nicht mehr verändert wird.

Wenn sie in der Klasse vorhanden ist, muss sie nicht mehr zur Laufzeit berechnet werden.

Generics

(Parametrischer Polymorphismus)

Wiederholung Datenstrukturen

Telefonbuchverwaltung mit einer einfachverketteten Liste



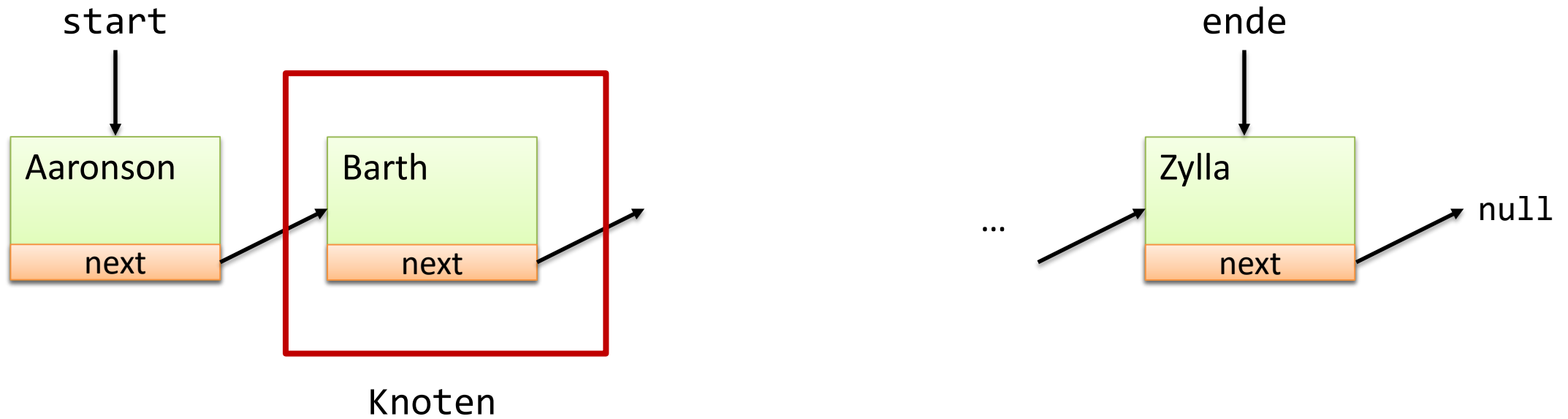
Was waren die Vorteile einer Liste?

Man kann ein Element an beliebiger Stelle...

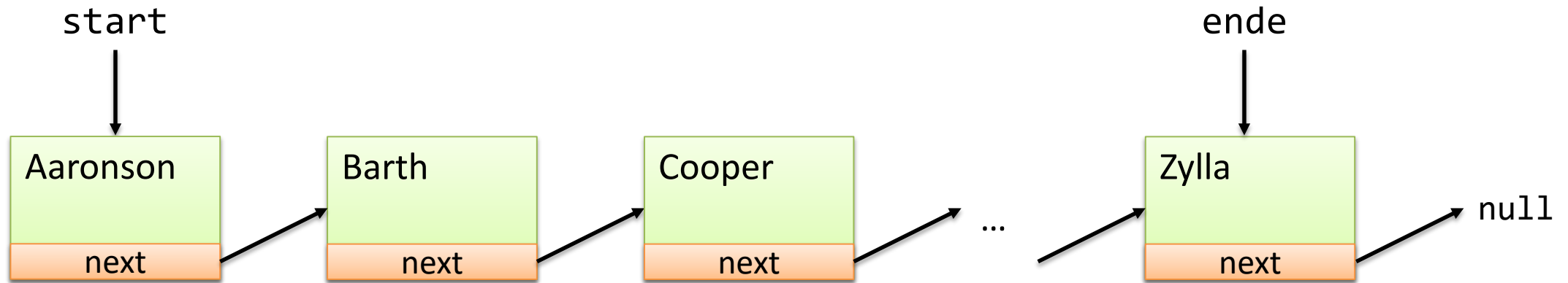
- ...in eine Liste einfügen
- ...aus der Liste löschen

Die Größe der Liste musste vorher nicht festgelegt werden.
Es gibt keine Lücken wie im Array.

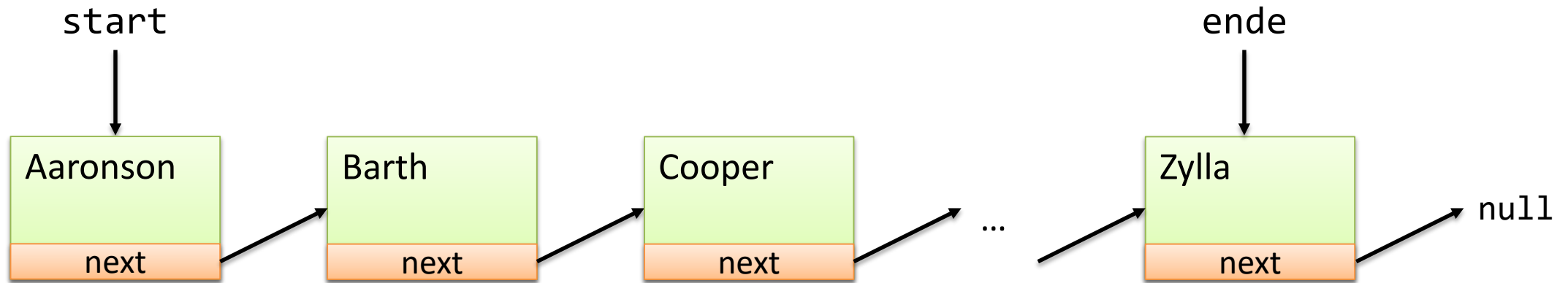
Telefonbuchverwaltung mit einer einfachverketteten Liste



Neue Einträge einfügen



Einträge löschen



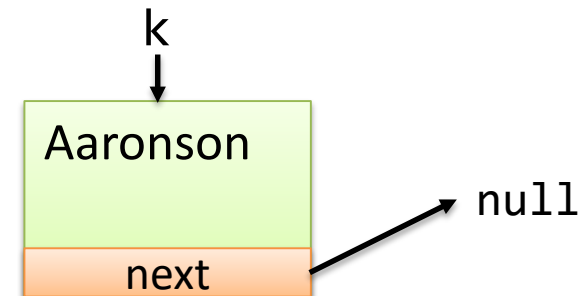
Wie war die Liste aufgebaut?

```
public class Liste
{
    Knoten start;
    Knoten ende;
    ...
    public void add(Person element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
        }
    }
}
```

```
public class Knoten
{
    Person element;
    Knoten next;

    public Knoten(Person element)
    {
        this.element = element;
    }

    public Person getElement()
    {
        return element;
    }
}
```



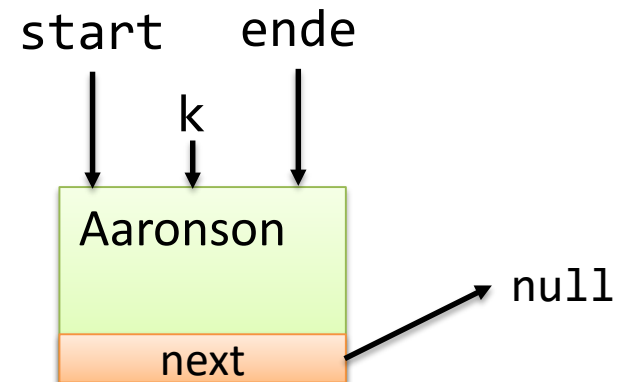
Wie war die Liste aufgebaut?

```
public class Liste
{
    Knoten start;
    Knoten ende;
    ...
    public void add(Person element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
    }
}
```

```
public class Knoten
{
    Person element;
    Knoten next;

    public Knoten(Person element)
    {
        this.element = element;
    }

    public Person getElement()
    {
        return element;
    }
}
```



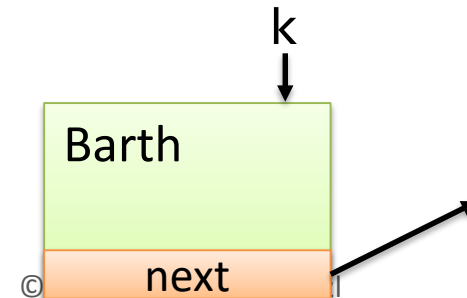
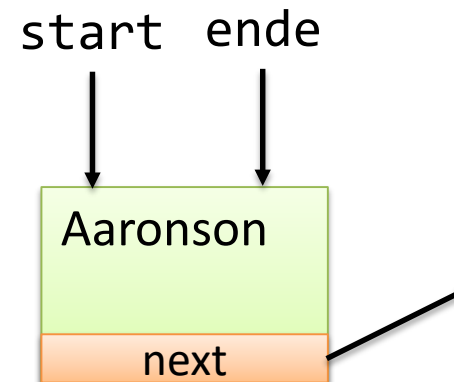
Wie war die Liste aufgebaut?

```
public class Liste
{
    Knoten start;
    Knoten ende;
    ...
    public void add(Person element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
}
```

```
public class Knoten
{
    Person element;
    Knoten next;

    public Knoten(Person element)
    {
        this.element = element;
    }

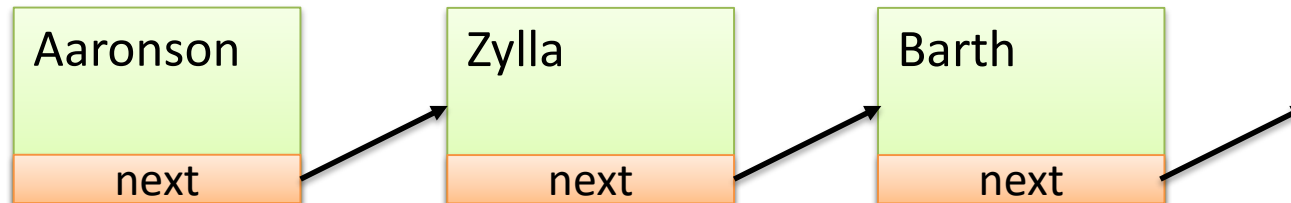
    public Person getElement()
    {
        return element;
    }
}
```



Wie haben wir die Liste genutzt?

```
public class Main
{
    public static void main(String[] args)
    {
        Liste personen = new Liste();
        personen.add(new Person("Aaronson"));
        personen.add(new Person("Zylla"));
        personen.add(new Person("Barth"));

        Person p = personen.removeFirst();
        System.out.println(p);
    }
}
```



Generics

(Parametrischer Polymorphismus)

Nehmen wir an, wir wollen die Liste auch für andere Datentypen verwenden als für **Person**, z.B. für **Stuhl**.

```
public class Liste
{
    Knoten start;
    Knoten ende;
    ...
    public void add(Person element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
}
```

```
public class Knoten
{
    Person element;
    Knoten next;

    public Knoten(Person element)
    {
        this.element = element;
    }

    public Person getElement()
    {
        return element;
    }
}
```

Nehmen wir an, wir wollen die Liste auch für andere Datentypen verwenden als für **Person**, z.B. für **Stuhl**.

```
public class Liste
{
    Knoten start;
    Knoten ende;
    ...
    public void add(Stuhl element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
}
```

```
public class Knoten
{
    Stuhl element;
    Knoten next;

    public Knoten(Stuhl element)
    {
        this.element = element;
    }

    public Stuhl getElement()
    {
        return element;
    }
}
```

➤ Wir brauchen für jeden Datentypen zwei eigene Klasse.

Erste Abhilfe durch Generalisierung

```
public class Liste
{
    Knoten start;
    Knoten ende;
    ...
    public void add(Object element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
}
```

```
public class Knoten
{
    Object element;
    Knoten next;

    public Knoten(Object element)
    {
        this.element = element;
    }

    public Object getElement()
    {
        return element;
    }
}
```

- Das sieht gut aus. Werfen wir einen Blick auf die Verwendung der Klasse.


```
public class Main
{
    public static void main(String[] args)
    {
        Liste personen = new Liste();
        personen.add(new Person("Aaronson"));
        personen.add(new Person("Zylla"));
        personen.add(new Person("Barth"));


        Person p = (Person) personen.removeFirst();
        System.out.println(p);
    }
}
```

➤ Das sieht auch gut aus. Aber...

Wäre es nicht gut,
die Typangabe einfach mitzugeben?

```
public static void main(String[] args)
{
    ...
    List files = new ArrayList();
    List urls = new ArrayList();
    List filenames = new ArrayList();


    FileReader fr = new FileReader(new File(...));
    LineNumberReader f = new LineNumberReader(fr);
    for (String line; (line = f.readLine()) != null;)
    {
        urls.add(new URL(line));
        filenames.add(line.substring(
            line.lastIndexOf('/') + 1));
        ...
        files.add(new File(line));
    }
    f.close();
    ...
    URL url = (URL) urls.remove(0);
    File filename = (File) filenames.remove(0);
    String file = (String) files.remove(0);
    ...
}
```



```
public static void main(String[] args)
{
    ...
    List<File> files = new ArrayList<File>();
    List<URL> urls = new ArrayList<URL>();
    List<String> filenames = new ArrayList<String>();


    FileReader fr = new FileReader(new File(...));
    LineNumberReader f = new LineNumberReader(fr);
    for (String line; (line = f.readLine()) != null;)
    {
        urls.add(new URL(line));
        filenames.add(line.substring(
            line.lastIndexOf('/') + 1));
        ...
        files.add(new File(line));
    }
    f.close();
    ...
    URL remove = urls.remove(0);
    File file = files.remove(0);
    String filename = filenames.remove(0);
    ...
}
```

```
53 URL url = (URL) urls.remove(0);  
54 File filename = (File) filenames.remove(0);  
55 String file = (String) files.remove(0);
```

 Laufzeitfehler

Bei einer Typisierung findet der Compiler bereits die Fehler:

```
29 URL remove = urls.remove(0);  
30 File filename = filenames.remove(0);  
31 String file = files.remove(0);
```



Compiler-Fehler

➤ Code wird schlanker und Fehler schon zur Compile-Time gefunden.

Zurück zu unserer Liste...

Unsere Liste war auf einen Datentypen zugeschnitten.

Wie kann man für unsere Liste das Verhalten aus gerade
gesehenem Beispiel implementieren?

Wir benötigen eine generische Klasse!

```

public class Liste<E>
{
    Knoten start;
    Knoten ende;
    ...
    public void add(E element)
    {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
}

```

- Eine **generische Klasse** ist eine durch einen **Typparameter** parametrisierbare Klasse.
 - Der Typparameter muss in spitzen Klammern hinter den Klassennamen geschrieben werden.
 - Als Buchstaben werden i.d.R.
 - E für Element
 - T für Typ
 - K für Key
 - V für Value
 - N für Number
 verwendet.
 - Mehrere Typparameter sind auch möglich, z.B. <T1,T2>
 - Als Typparameter sind nur Referenztypen zulässig.
- Eine generische Klasse ermöglicht es eine Klasse **einmal** zu implementieren...

- ...und auf **verschiedene konkrete Datentypen** anzuwenden.

```
public class Main
{
    public static void main(String[] args)
    {
        Liste<Person> personen = new Liste<Person>();
        personen.add(new Person("Aaronson"));
        personen.add(new Person("Zylla"));
        personen.add(new Person("Barth"));

        Person p = personen.removeFirst();
        System.out.println(p);
    }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
        Liste<Stuhl> stuehle = new Liste<Stuhl>();
        stuehle.add(new Stuhl(1));
        stuehle.add(new Stuhl(2));
        stuehle.add(new Stuhl(3));

        Stuhl s = stuehle.removeFirst();
        System.out.println(s);
    }
}
```

Eine Kurzschreibweise ist übrigens auch möglich:

```
Liste<Person> personen = new Liste<Person>();
```



```
Liste<Person> personen = new Liste<>();
```

Der Compiler kann das richtige Typargument herleiten.

Wir möchten ein Objekt in eine (nicht-generische) Liste einfügen!

```
35 List files = new ArrayList();  
36 List urls = new ArrayList();  
37 List filenames = new ArrayList();  
38  
39 urls.add(new URL(line));  
40 filenames.add(new File(filename));  
41 files.add(filename);
```


Wir möchten ein Objekt in eine (nicht-generische) Liste einfügen!

```
34  
35 List files = new ArrayList();  
36 List urls = new ArrayList();  
37 List filenames = new ArrayList();  
38  
39 urls.add(new URL(line));  
40 filenames.add(new File(filename));  
41 files.add(filename);
```

Type safety: The method add(Object) belongs to the raw type List. References to generic type List<E> should be parameterized



Möglicher Fehler wird erst zur Laufzeit erkannt!

Wir möchten ein Objekt in eine (generische) Liste einfügen!

```
18 List<File> files = new ArrayList<>();  
19 List<URL> urls = new ArrayList<>();  
20 List<String> filenames = new ArrayList<>();  
21  
22 urls.add(new URL(line));  
23 filenames.add(new File(filename));  
24 files.add(filename);
```

The method add(File) in the type List<File> is not applicable for the arguments (String)

Möglicher Fehler wird schon zur Compile-Time erkannt!

Wir würden Sie folgende **removeFirst**-Implementierung
für unsere generische Liste anpassen?

```
public class Liste<E> {
    Knoten start;
    Knoten ende;
    ...
    public void add(E element) {
        if (element == null) return;
        Knoten k = new Knoten(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
    public E removeFirst() {
        Knoten temp = start;
        start = start.next;
        return temp.getElement();
    }
}
```

A:

```
public Object removeFirst() {
    Knoten temp = start;
    start = start.next;
    return temp.getElement();
}
```

B:

```
public E removeFirst() {
    Knoten temp = start;
    start = start.next;
    return temp.getElement();
}
```

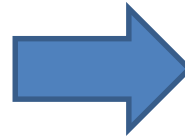
C:

```
public E removeFirst() {
    Knoten temp = start;
    start = start.next;
    return (E) temp.getElement();
}
```

D:

```
public Object removeFirst() {
    Knoten temp = start;
    start = start.next;
    return (Object) temp.getElement();
}
```

```
public E removeFirst() {  
    Knoten temp = start;  
    start = start.next;  
    return (E) temp.getElement();  
}
```



```
public E removeFirst() {  
    Knoten temp = start;  
    start = start.next;  
    return temp.getElement();  
}
```

Wie können wir den Cast auf E loswerden?

Wir machen aus Knoten auch eine generische Klasse!

```

public class Liste<E> {
    Knoten<E> start;
    Knoten<E> ende;
    ...
    public void add(E element) {
        if (element == null) return;
        Knoten<E> k = new Knoten<>(element);
        if (start == null)
        {
            start = k;
            ende = start;
        }
        else
        {
            ende.next = k;
            ende = k;
        }
    }
    public E removeFirst() {
        Knoten<E> temp = start;
        start = start.next;
        return temp.getElement();
    }
}

```

```

public class Knoten<E>
{
    E element;
    Knoten<E> next;

    public Knoten(E element)
    {
        this.element = element;
    }

    public E getElement()
    {
        return element;
    }
}

```

Weiteres Beispiel: Paare

- Handschuhe sollten immer paarweise auftreten.
- Ein Handschuh könnte folgendermaßen aussehen:



```
public class Glove
{
    boolean fingerless;

    public Glove(boolean fingerless)
    {
        this.fingerless = fingerless;
    }

    public String toString()
    {
        if (fingerless)
            return "fingerloser Handschuh";
        else return "Handschuh";
    }
}
```

Weiteres Beispiel: Paare

- Eine Klasse zur Verwaltung von Handschuhpaaren könnte wie folgt aussehen:



```
public class PairOfGloves {  
    private Glove left;  
    private Glove right;  
  
    public PairOfGloves(Glove l, Glove r) {  
        this.left = l;  
        this.right = r;  
    }  
  
    public Glove getLeft() {  
        return left;  
    }  
  
    public Glove getRight() {  
        return right;  
    }  
  
    public String toString() {  
        return "(l=" + left.toString() +  
            ", r=" + right.toString() + ")";  
    }  
}
```

Weiteres Beispiel: Paare

- Verwendung:



```
public static void main(String[] args)
{
    Glove g1 = new Glove(true);
    Glove g2 = new Glove(true);

    PairOfGloves pair = new PairOfGloves(g1, g2);
    System.out.println(pair.toString());

    Glove left = pair.getLeft();
    System.out.println(left.toString());
}
```

```
C: />java PairOfGlovesTest.class
(l=fingerloser Handschuh, r=fingerloser Handschuh)
fingerloser Handschuh
C: />
```

Weiteres Beispiel: Paare

- Socken sollten auch immer paarweise auftreten.



```
public class Sock
{
    int size;

    public Sock(int size)
    {
        this.size = size;
    }

    public String toString()
    {
        return "Sock der Größe " + size;
    }

    public int getSize()
    {
        return size;
    }
}
```


Weiteres Beispiel: Paare

- Wie kann man die PairOfGloves Klasse auch für die Socken verwenden?



```
public class PairOfGloves {  
    private Glove left;  
    private Glove right;  
  
    public PairOfGloves(Glove l, Glove r) {  
        this.left = l;  
        this.right = r;  
    }  
  
    public Glove getLeft() {  
        return left;  
    }  
  
    public Glove getRight() {  
        return right;  
    }  
  
    public String toString() {  
        return "(l=" + left.toString() +  
            ", r=" + right.toString() + ")";  
    }  
}
```

Weiteres Beispiel: Paare

- Wie kann man die PairOfGloves auch für die Socken verwenden?
- Glove mit Sock ersetzen?
- extra Klasse



```
public class PairOfSocks {  
    private Sock left;  
    private Sock right;  
  
    public PairOfSocks(Sock l, Sock r) {  
        left = l;  
        right = r;  
    }  
  
    public Sock getLeft() {  
        return left;  
    }  
  
    public Sock getRight() {  
        return right;  
    }  
  
    public String toString() {  
        return "(l=" + left.toString() +  
            ", r=" + right.toString() + ")";  
    }  
}
```

Weiteres Beispiel: Paare

- Wie kann eine Klasse beides unterstützen?
- Generalisierung?



```
public class PairOfObjects {  
    private Object left;  
    private Object right;  
  
    public PairOfObjects(Object l, Object r) {  
        left = l;  
        right = r;  
    }  
  
    public Object getLeft() {  
        return left;  
    }  
  
    public Object getRight() {  
        return right;  
    }  
  
    public String toString() {  
        return "(l=" + left.toString() +  
            ", r=" + right.toString() + ")";  
    }  
}
```

Weiteres Beispiel: Paare

Problem:

- Downcast ist notwendig, um an die Informationen des Socks zu kommen



```
public static void main(String[] args)
{
    Sock s1 = new Sock(45);
    Sock s2 = new Sock(45);

    PairOfObjects pair = new PairOfObjects(s1, s2);

    //Downcast
    Sock left = (Sock) pair.getLeft();
    System.out.println(left.getSize());
}
```

```
C: />j ava PairOfObjectsTest1.class
45
C: />
```


Weiteres Beispiel: Paare

Problem:

- Socken und Handschuhe können zusammen Paare bilden.



```
public static void main(String[] args)
{
    Sock s1 = new Sock(45);
    Glove g1 = new Glove(true);

    PairOfObjects pair = new PairOfObjects(s1, g1);
    System.out.println(pair.toString());
}
```

```
C: />java PairOfObjectsTest2.class
(l=Sock der Größe 45, r=fingerloser Handschuh)
C: />
```

Weiteres Beispiel: Paare

- Wie kann eine Klasse beides unterstützen?
- Parametrisierbare (generische) Klasse



```
public class Pair<T> {  
    private T left;  
    private T right;  
  
    public Pair(T l, T r) {  
        left = l;  
        right = r;  
    }  
  
    public T getLeft() {  
        return left;  
    }  
  
    public T getRight() {  
        return right;  
    }  
  
    public String toString() {  
        return "(l=" + left.toString()  
            + ", r=" + right.toString() + ")";  
    }  
}
```

Weiteres Beispiel: Paare

Kein Downcast erforderlich!



```
public static void main(String[] args)
{
    Sock s1 = new Sock(45);
    Sock s2 = new Sock(45);

    Pair<Sock> pair = new Pair<Sock>(s1, s2);

    Sock left = pair.getLeft();
    System.out.println(left.getSize());
}
```

```
C: />java PairTest1.class
45
C: />
```

Weiteres Beispiel: Paare

The constructor Pair<Sock>(Sock, Glove) is undefined

```
Sock s1 = new Sock(45);  
Glove g1 = new Glove(true);  
Pair<Sock> pair = new Pair<Sock>(s1, g1);  
System.out.println(pair.toString());
```

Compiler findet “Fehler” zur Übersetzungszeit!



Falls es wirklich gewünscht wird, Socken und Handschuhe als Paar zu mischen:

```
Pair<Object> pair = new Pair<Object>(s1, g1);
```


Einschränkung von Typparametern

```
public class HalfOfPair
{
    protected boolean left;

    public boolean isLeft() {
        return left;
    }


    public boolean isRight() {
        return !left;
    }
}
```

```
public class Pair<T extends HalfOfPair>
{
    T left;
    T right;

    public Pair(T l, T r)
    {
        if (l.isLeft()) left = l;
        if (r.isRight()) right = r;
    }
}
```

Pair lässt sich nur mit Subtypen (Unterklassen) von HalfOfPair instantiieren!

```
public class Glove extends HalfOfPair
{
    ...
}
```



```
public class Sock extends HalfOfPair
{
    ...
}
```

Literatur



<https://docs.oracle.com/javase/tutorial/java/generics/index.html>



S. Heinzl, M. Mathes: *Middleware in Java*, Vieweg+Teubner, ISBN 3-528-05912-5, 2005



R. Schiedermeier: *Programmieren mit Java*, Pearson Studium; Auflage: 2., aktualisierte Auflage, ISBN 3868940316, 2010

© Prof. Dr. Steffen Heinzl