*Sommersemester 2020*

CARL VON OSSIETZKY universität OLDENBURG

# Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

# Input/Output

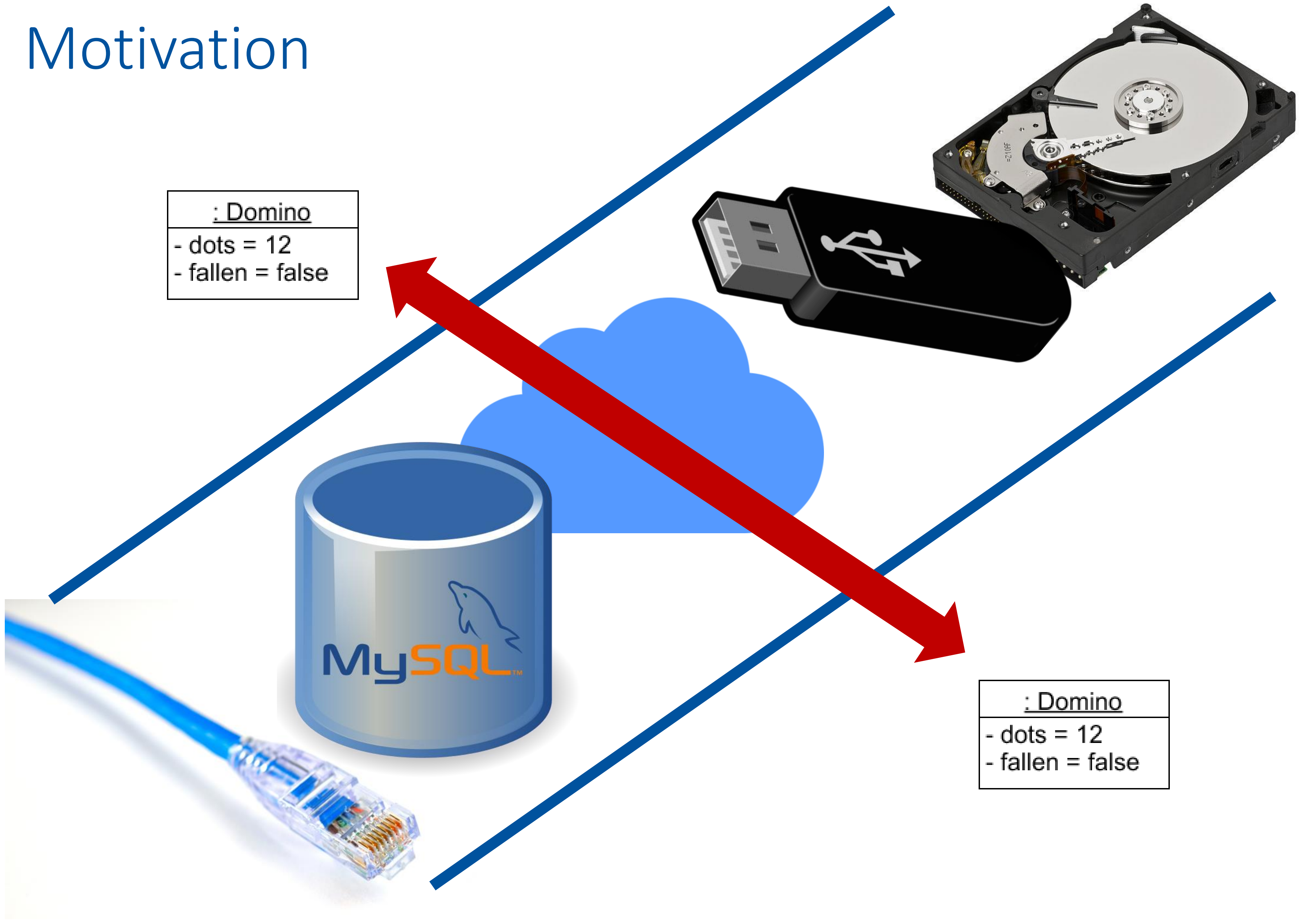
## Lernziele

- **InputStream**, **OutputStream**
- Decorator-Pattern
- **Serializable**
- **Path**, **File**

## Motivation

- Bisher: Daten werden angelegt, wo sie gebraucht werden
- Realität: Daten werden irgendwo angelegt, gespeichert, verschickt, geladen, verändert, direkt verschickt, wieder verändert, usw.
- Aufgabe:
  - Daten aus dem Arbeitsspeicher in ein beschreibbares „Etwas“ (Festplatte, USB-Stick, Cloud, Netzwerk, …) schreiben und wieder zurück lesen
  - Java-internes Datenformat in eine geeignete Form serialisieren (= Umwandlung von strukturierten Daten in eine sequentielle Form)

# Motivation



: Domino
- dots = 12
- fallen = false

: Domino
- dots = 12
- fallen = false

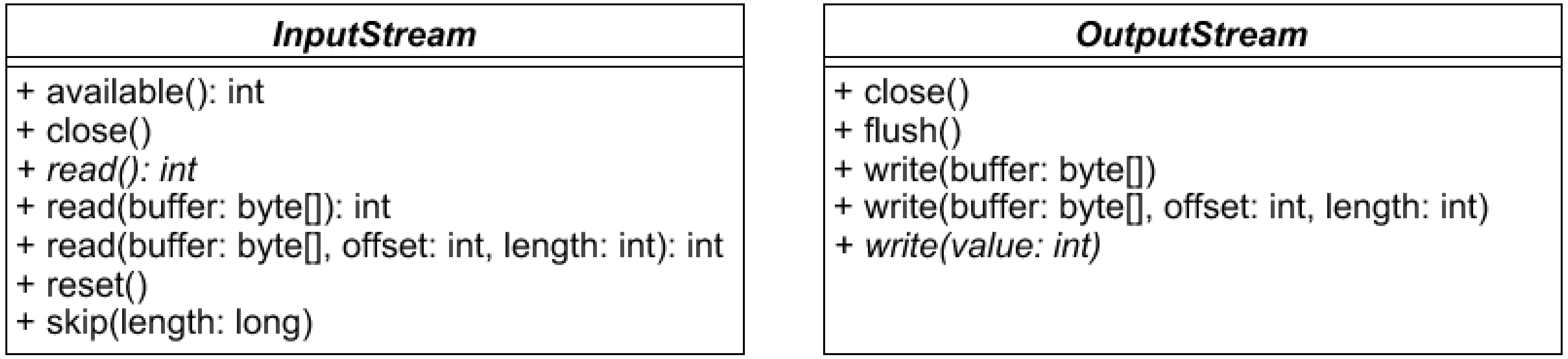
CARL VON OSSIETZKY universität OLDENBURG

## I/O

# I/O Streams

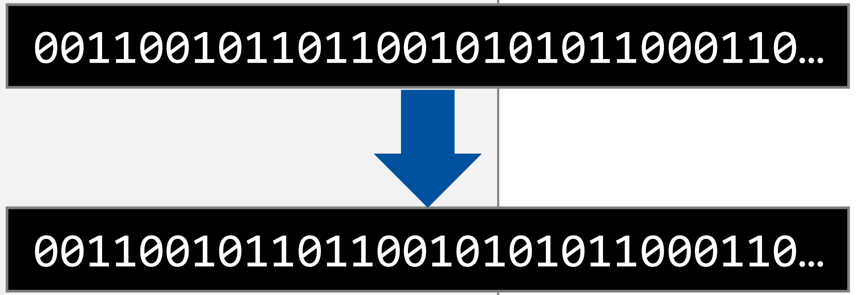
- Abstraktion des o.g. „Etwas“ in das man schreiben bzw. von dem man lesen kann
  - **`InputStream`** → Eingabe, Lesen
  - **`OutputStream`** → Ausgabe, Schreiben
  - nicht zu verwechseln mit dem Java 8-Interface **`Stream<T>`**
- Datenstrom, Byte-basiert
  - Lesen/Schreiben eines Bytes und verwandte Methoden
  - Spezialisierungen, die das Lesen/Schreiben von primitiven Datentypen (**`int`**, **`boolean`**, **`double`**, …) und sogar Objekten erlauben

# I/O Streams

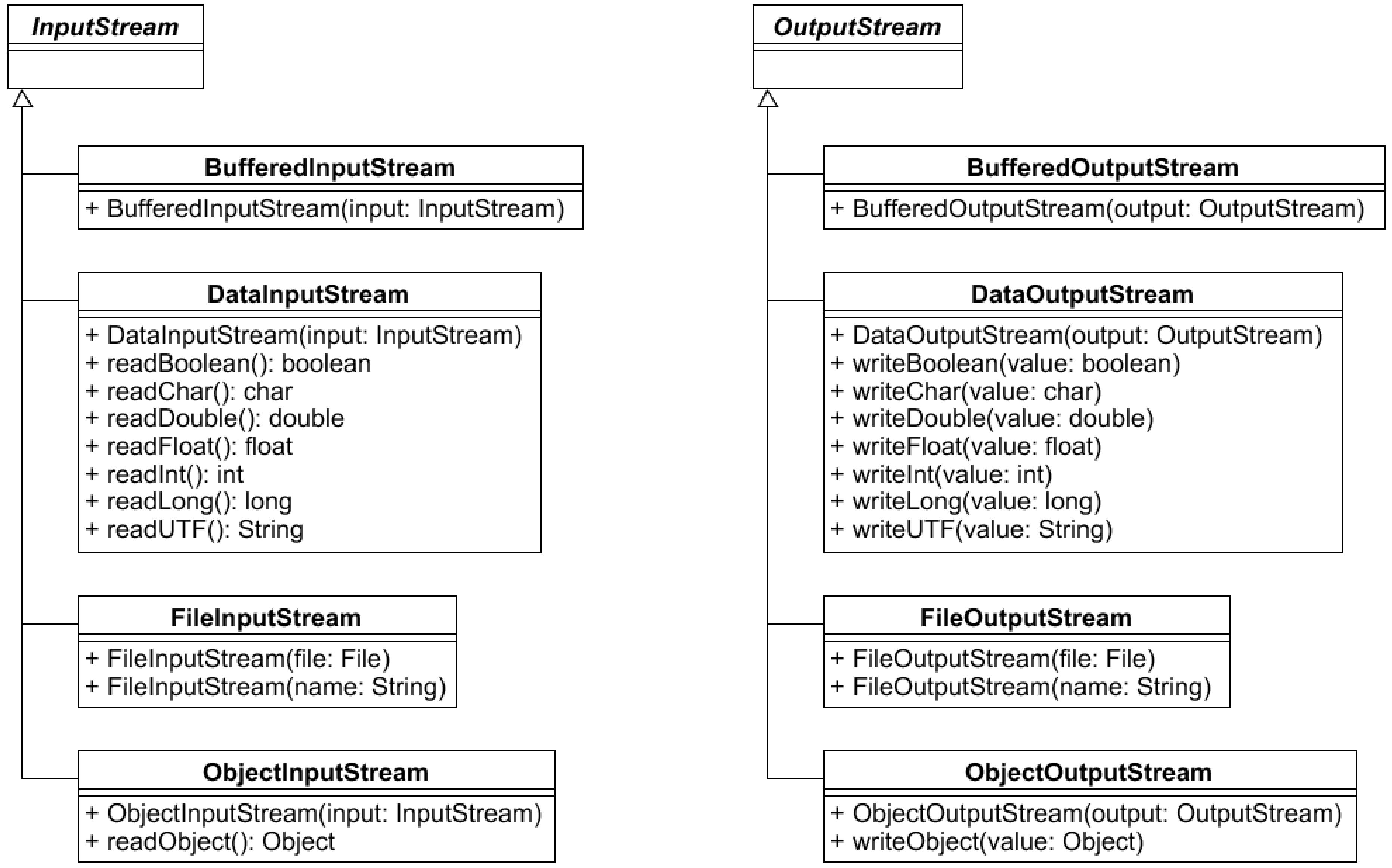

*InputStream*
+ available(): int
+ close()
+ *read(): int*
+ read(buffer: byte[]): int
+ read(buffer: byte[], offset: int, length: int): int
+ reset()
+ skip(length: long)
*OutputStream*
+ close()
+ flush()
+ write(buffer: byte[])
+ write(buffer: byte[], offset: int, length: int)
+ *write(value: int)*

## I/O Streams

```java
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        InputStream in = null;
        OutputStream out = null;
        try {
            in = …     ?
            out = …
            int c = in.read();
            while (c != -1) {
                out.write(c);
                c = in.read();
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    } }
```



# I/O Streams (2)



*InputStream*
BufferedInputStream
+ BufferedInputStream(input: InputStream)
DataInputStream
+ DataInputStream(input: InputStream)
+ readBoolean(): boolean
+ readChar(): char
+ readDouble(): double
+ readFloat(): float
+ readInt(): int
+ readLong(): long
+ readUTF(): String
FileInputStream
+ FileInputStream(file: File)
+ FileInputStream(name: String)
ObjectInputStream
+ ObjectInputStream(input: InputStream)
+ readObject(): Object
*OutputStream*
BufferedOutputStream
+ BufferedOutputStream(output: OutputStream)
DataOutputStream
+ DataOutputStream(output: OutputStream)
+ writeBoolean(value: boolean)
+ writeChar(value: char)
+ writeDouble(value: double)
+ writeFloat(value: float)
+ writeInt(value: int)
+ writeLong(value: long)
+ writeUTF(value: String)
FileOutputStream
+ FileOutputStream(file: File)
+ FileOutputStream(name: String)
ObjectOutputStream
+ ObjectOutputStream(output: OutputStream)
+ writeObject(value: Object)

## I/O Streams (2)

```
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        InputStream in = null;
        OutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c = in.read();
            while (c != -1) {
                out.write(c);
                c = in.read();
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    } }
```

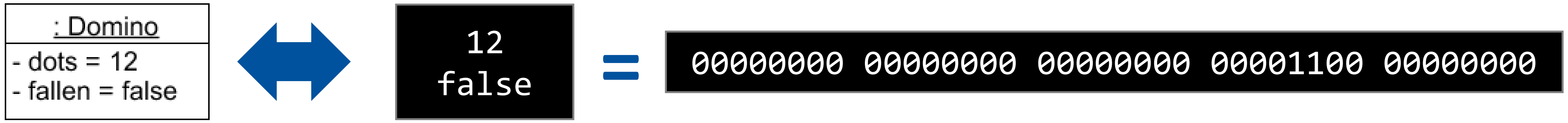
## I/O Streams (3)

```java
public class CopyBytes {
    public static void main(String[] args) throws IOException {
        InputStream in = null;
        OutputStream out = null;
        try {
            in = new BufferedInputStream(
                           new FileInputStream("input.txt"));
            out = new BufferedOutputStream(
                           new FileOutputStream("output.txt"));
            int c = in.read();
            while (c != -1) {
                out.write(c);
                c = in.read();
            }
        } finally {
            …
        }
    }
}
```

## Dominos speichern

```
public class Domino {
    private int dots;
    private boolean fallen;

    public Domino(int dots) {
        this.dots = dots;
    }

    public static void save(String filename, Domino piece) throws IOException {
        DataOutputStream out = null;
        try {
            out = new DataOutputStream(new BufferedOutputStream(
                                new FileOutputStream(filename)));
            out.writeInt(piece.getDots());
            out.writeBoolean(piece.isFallen());
        } finally {
            if (out != null) {
                out.close();
            }
        }
    }
```

# Dominos speichern



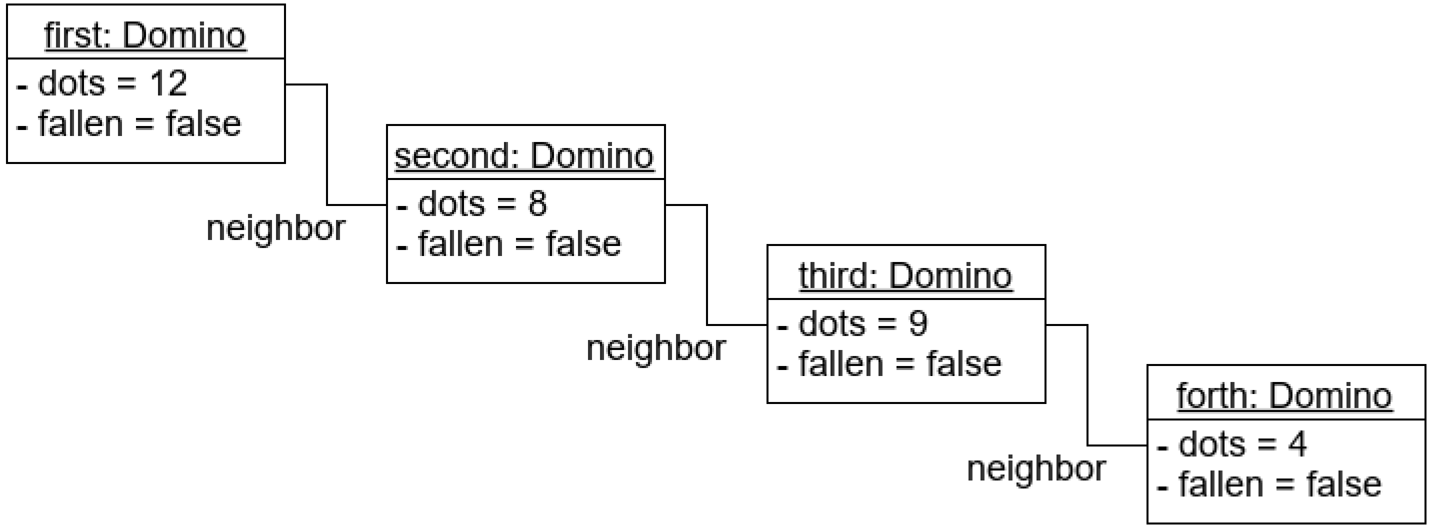
## Dominos laden

```
    public static Domino load(String filename) throws IOException {
        Domino piece = null;
        DataInputStream in = null;
        try {
            in = new DataInputStream(new BufferedInputStream(
                            new FileInputStream(filename)));
            int dots = in.readInt();
            piece = new Domino(dots);
            piece.setFallen(in.readBoolean());
        } finally {
            if (in != null) {
                in.close();
            }
        }
        return piece;
    }
```

# Dominos speichern (2)

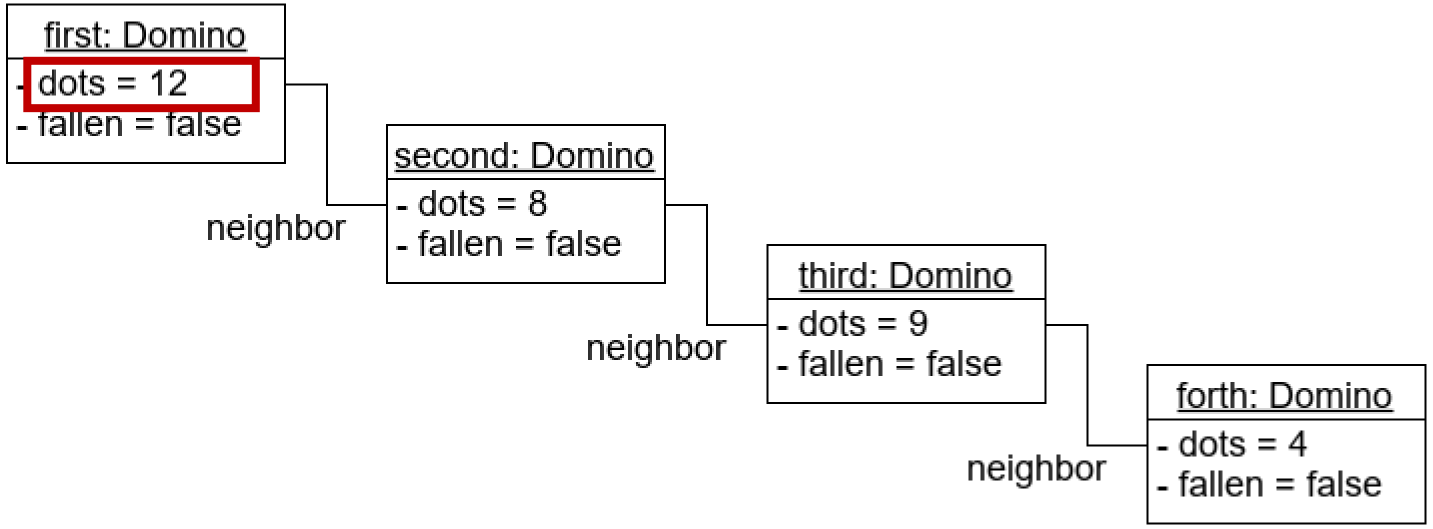
```
public static void save(String filename, Domino piece) throws IOException {
    DataOutputStream out = null;
    try {
        out = new DataOutputStream(new BufferedOutputStream(
                        new FileOutputStream(filename)));
        save(piece, out);
    } finally { … }
}

private static void save(Domino piece, DataOutputStream out) throws IOException {
    out.writeInt(piece.getDots());
    out.writeBoolean(piece.isFallen());
    if (piece.getNeighbor() == null) {
        out.writeBoolean(false);
    } else {
        out.writeBoolean(true);
        save(piece.getNeighbor(), out);
    }
}
```

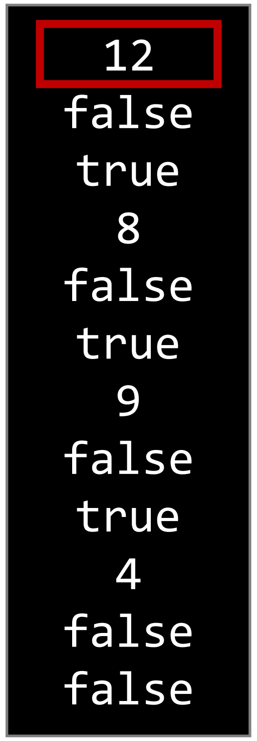
# Dominos speichern (2)



```
12
false
true
8
false
true
9
false
true
4
false
false
```

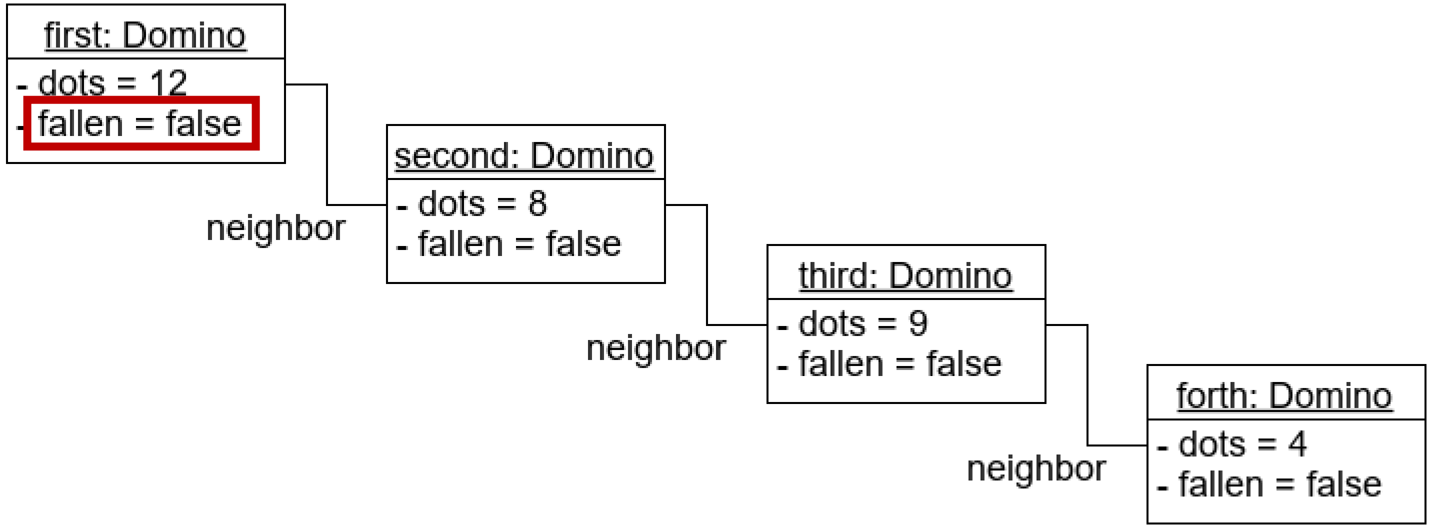
# Dominos speichern (2)

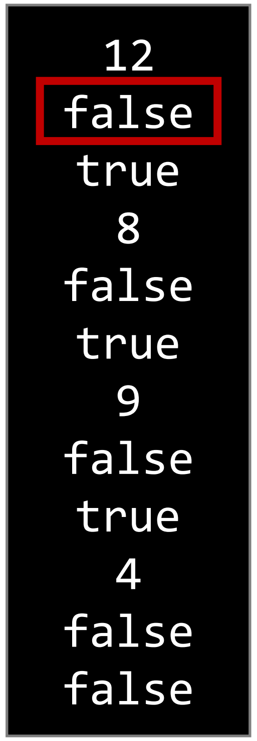




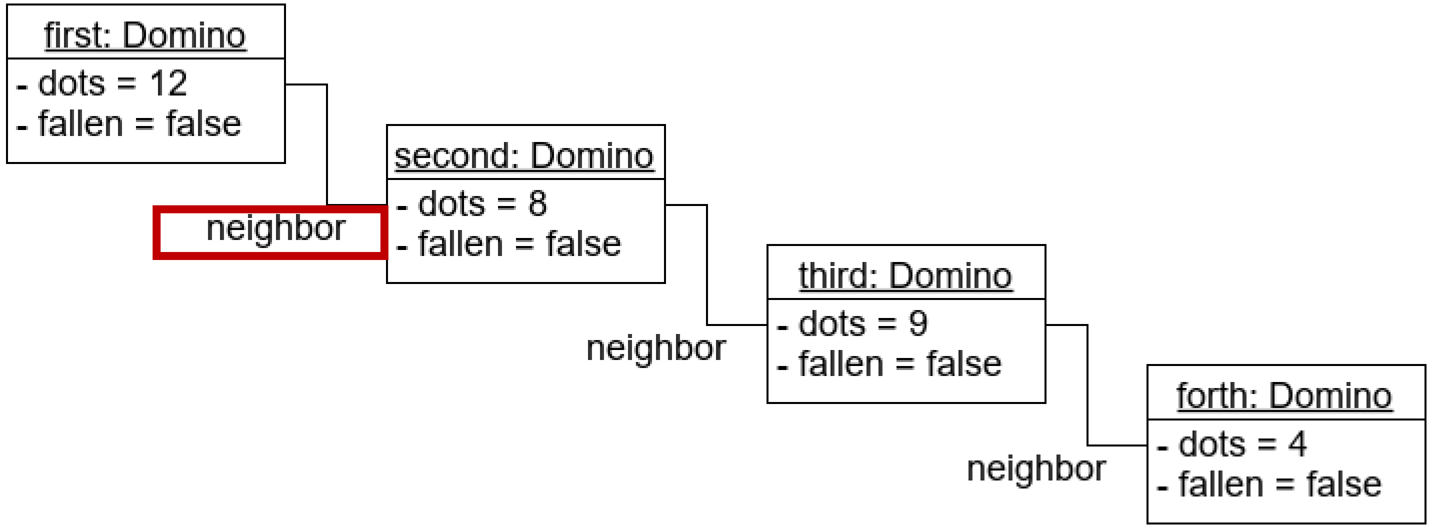
```
12
false
true
8
false
true
9
false
true
4
false
false
```

## Dominos speichern (2)



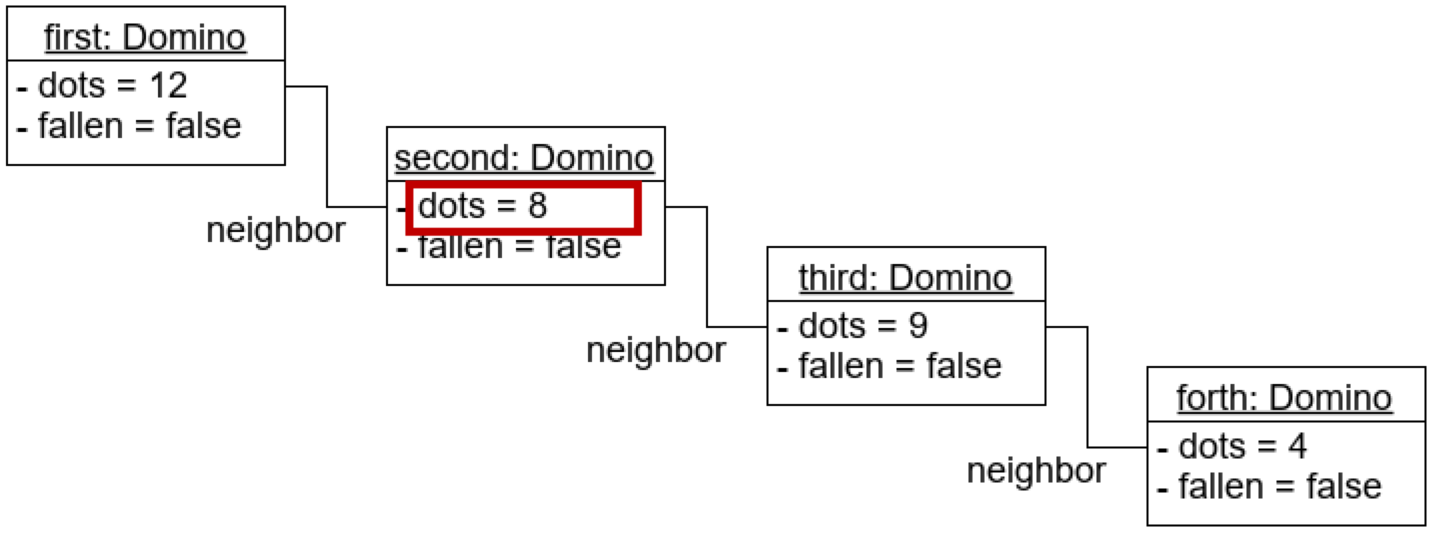

```
12
false
true
8
false
true
9
false
true
4
false
false
```

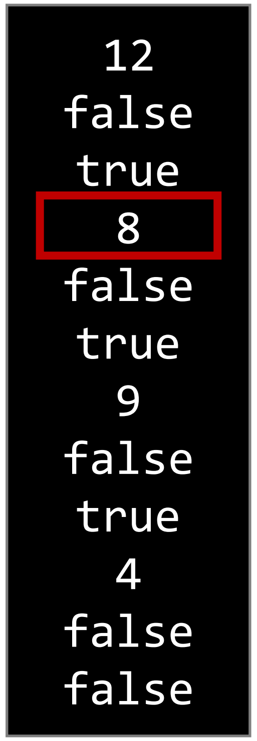
# Dominos speichern (2)



```
12
false
true
8
false
true
9
false
true
4
false
false
```

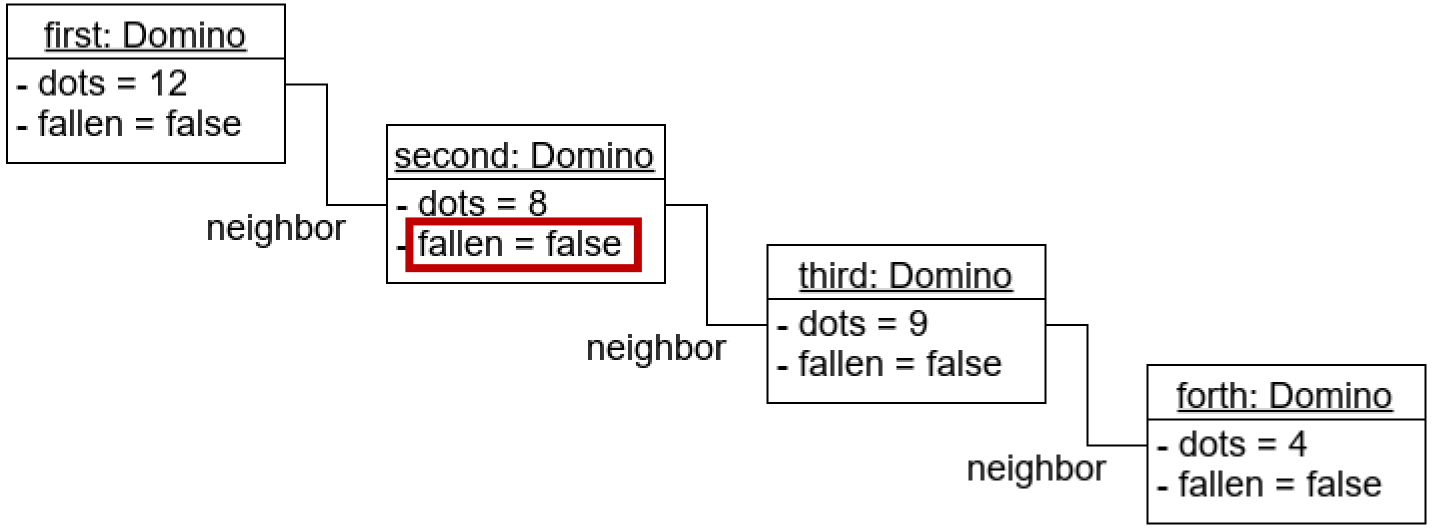
# Dominos speichern (2)

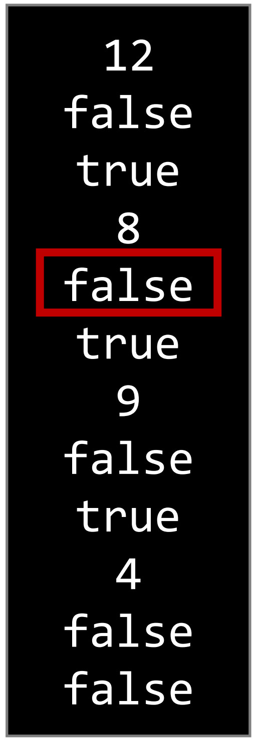




```
12
false
true
8
false
true
9
false
true
4
false
false
```

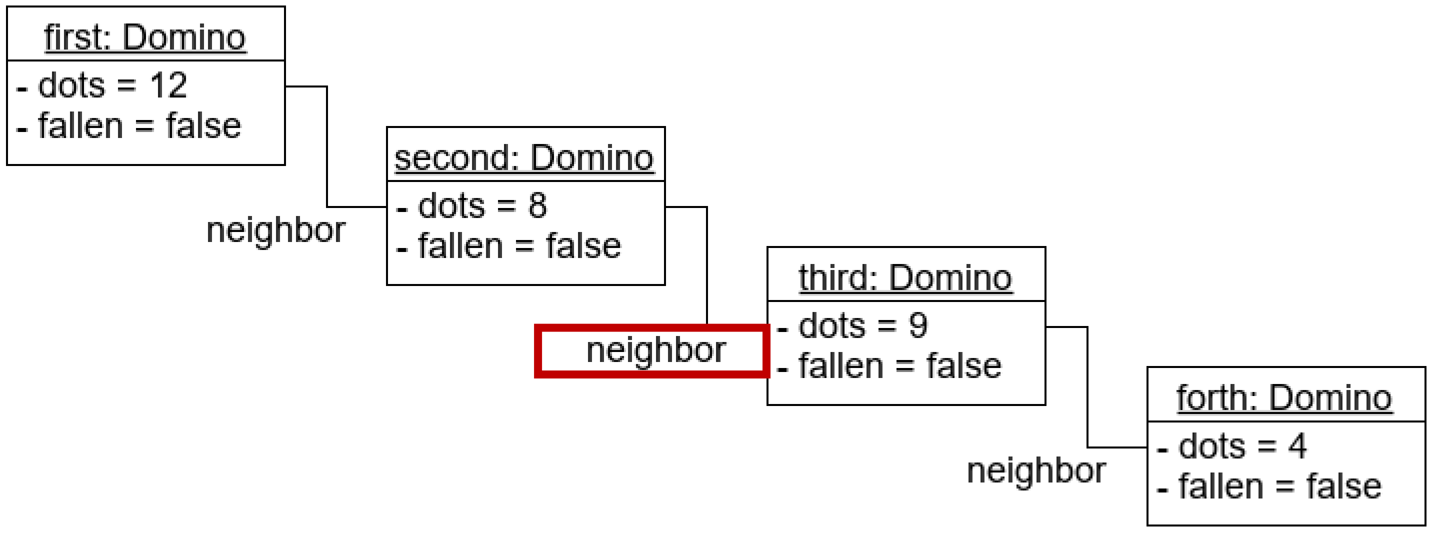
# Dominos speichern (2)





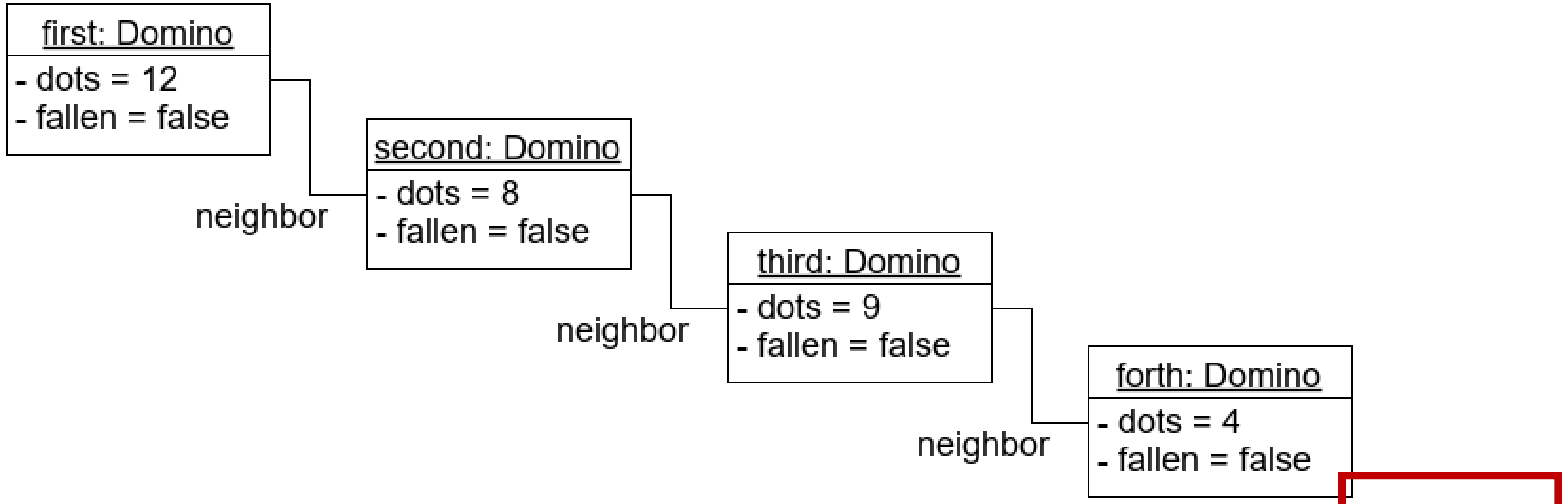
```
12
false
true
8
false
true
9
false
true
4
false
false
```

# Dominos speichern (2)



```
12
false
true
8
false
true
9
false
true
4
false
false
```

# Dominos speichern (2)



```
12
false
true
8
false
true
9
false
true
4
false
false
```

# Dominos speichern (2)

```java
Domino piece = new Domino();
piece.setNeighbor(piece);
Domino.save("domino.bin", piece);
```

```
Exception in thread "main" java.lang.StackOverflowError
at Domino.save(Domino.java:58)
at Domino.save(Domino.java:64)
at Domino.save(Domino.java:64)
at Domino.save(Domino.java:64)
at Domino.save(Domino.java:64)
…
```

Objektidentität geht verloren:
Nachbar wird als neue Kopie gespeichert.
Nachbar des Nachbarn wird als neue Kopie gespeichert.
Nachbar des Nachbarn des Nachbarn …
→ wir müssen uns merken, welche Objekte wir bereits gespeichert haben

## Objektidentität bewahren: Idee

- Jedes Objekt durch eine eindeutige ID repräsentieren
  - z.B. **`Domino.getId(): int`**
- Jedes Objekt nur einmal speichern/laden, danach auf die Referenz zugreifen
  - Speichern: IDs der bereits gespeicherten Objekte in einer Menge (**`Set<Integer>`**) merken, so dass kein Objekt doppelt gespeichert wird
    - wenn versucht wird, ein Objekt mehrfach zu speichern, ab dem zweiten Mal nur noch die ID speichern
  - Laden: bereits geladene Objekte in einer Map (**`Map<Integer, Object>`**) unter ihrer ID speichern
    - wenn versucht wird, ein Objekt mehrfach zu laden, ab dem zweiten Mal nur die ID laden und das Objekt in der Map nachschlagen

## Dominos speichern (3)

```
public class Domino {

    private int id;
    private int dots;
    private boolean fallen;
    private Domino neighbor;

    private static int numberOfPieces = 0;

    public Domino(int dots) {
        id = numberOfPieces;
        numberOfPieces++;
        this.dots = dots;
    }

```

# Dominos speichern (3)

```
public static void save(String filename, Domino piece)
            throws IOException {
    DataOutputStream out = null;
    try {
        out = new DataOutputStream(new BufferedOutputStream(
                        new FileOutputStream(filename)));
        save(piece, out, new HashSet<Integer>());
    } finally {
        if (out != null) {
            out.close();
        }
    }
}
```

## Dominos speichern (3)

```
private static void save(Domino piece, DataOutputStream out,
            Set<Integer> set) throws IOException {
    if (set.contains(piece.getId())) {
        out.writeBoolean(true);
        out.writeInt(piece.getId());
    } else {
        set.add(piece.getId());
        out.writeBoolean(false);
        out.writeInt(piece.getId());
        out.writeInt(piece.getDots());
        out.writeBoolean(piece.isFallen());
        if (piece.getNeighbor() == null) {
            out.writeBoolean(false);
        } else {
            out.writeBoolean(true);
            save(piece.getNeighbor(), out, set);
        }
    }
}
```

# Dominos laden (3)

```
public static Domino load(String filename) throws IOException {
    Domino piece = null;
    DataInputStream in = null;
    try {
        in = new DataInputStream(new BufferedInputStream(
                        new FileInputStream(filename)));
        piece = load(in, new HashMap<Integer, Domino>());
    } finally {
        if (in != null) {
            in.close();
        }
    }
    return piece;
}
```

# Dominos laden (3)

```
private static Domino load(DataInputStream in,
           Map<Integer, Domino> map) throws IOException {
    boolean isReference = in.readBoolean();
    Domino piece = null;
    if (isReference) {
        piece = map.get(in.readInt());
    } else {
        int id = in.readInt();
        int dots = in.readInt();
        piece = new Domino(dots);
        piece.setFallen(in.readBoolean());
        map.put(id, piece);
        boolean hasNeighbor = in.readBoolean();
        if (hasNeighbor) {
            piece.setNeighbor(load(in, map));
        }
    }
    return piece;
}
```

# try-with-resources

- Einem **`try`**-Block kann eine Instanz des **`AutoCloseable`**-Interfaces als Parameter übergeben werden

```
public class MyResource implements AutoCloseable {
    @Override
    public void close() throws Exception { … }
    …
}
```

```
try (MyResource res = new MyResource()) {
    res.read();
} catch (SomeException e) {
    e.printStackTrace();
}
```

- Die **`close()`**-Methode wird von Java automatisch aufgerufen, es ist also kein **`finally`**-Block nötig

## try-with-resources: Beispiel

```
public class Domino {
    private int dots;
    private boolean fallen;
    public Domino(int dots) {
        this.dots = dots;
    }
    public static void save(String filename, Domino piece)
                throws IOException {
        try (DataOutputStream out = new DataOutputStream(
                    new BufferedOutputStream(
                            new FileOutputStream(filename)))) {
            out.writeInt(piece.getDots());
            out.writeBoolean(piece.isFallen());
        }
    }
}
```

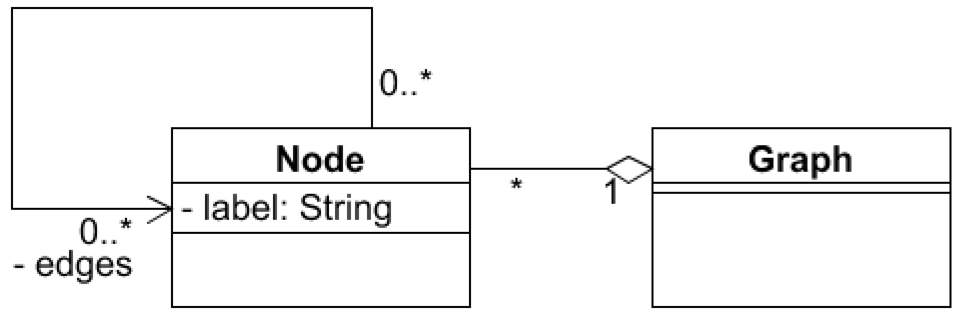
## try-with-resources: Beispiel

```java
public static Domino load(String filename) throws IOException {
    Domino piece = null;
    try (DataInputStream in = new DataInputStream(
                new BufferedInputStream(
                        new FileInputStream(filename)))) {
        int dots = in.readInt();
        piece = new Domino(dots);
        piece.setFallen(in.readBoolean());
    }
    return piece;
}
```

**`InputStream`** und **`OutputStream`** implementieren beide **`AutoCloseable`**.
Ihre **`close()`**-Methoden können eine **`IOException`** werfen,
die nach wie vor behandelt werden muss.

# Wiederholung: Graphen

```
public class Graph {

    private List<Node> nodes;

}

class Node {

    private String label;
    private List<Node> edges;

}
```



## Graphen speichern

```java
public static void save(String filename, Graph graph) throws IOException {
    try (DataOutputStream out = new DataOutputStream(new BufferedOutputStream(
                new FileOutputStream(filename)))) {
        Map<Node, Integer> nodeIdMap = new HashMap<>();
        int id = 0;
        out.writeInt(graph.getNodes().size());
        for (Node node : graph.getNodes()) {
            nodeIdMap.put(node, id);
            out.writeInt(id);
            id++;
        }
        for (Node node : graph.getNodes()) {
            out.writeUTF(node.getLabel());
            out.writeInt(node.getEdges().size());
            for (Node edge : node.getEdges()) {
                out.writeInt(nodeIdMap.get(edge));
            }
        }
    }
}
```

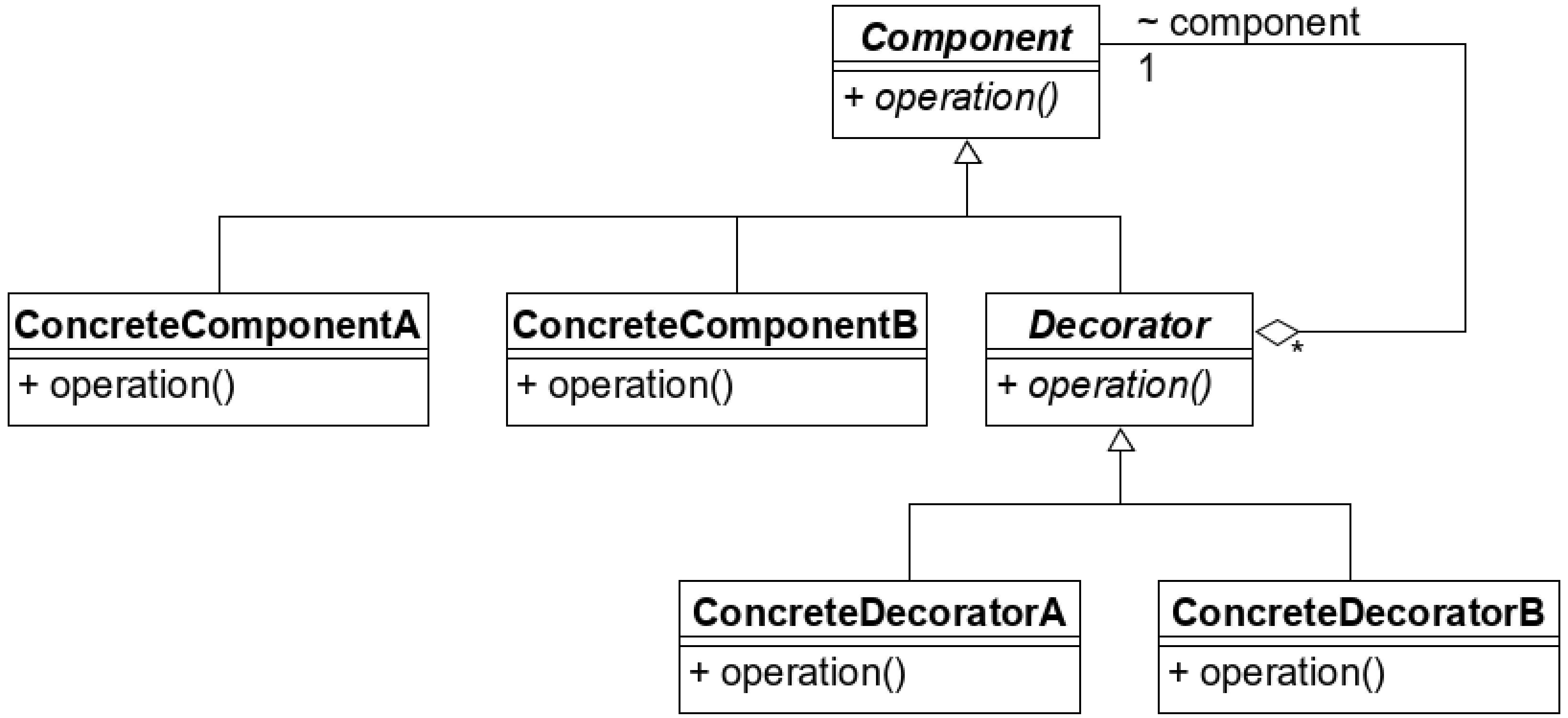
# Graphen laden

```java
public static Graph load(String filename) throws IOException {
    Graph graph = null;
    try (DataInputStream in = new DataInputStream(new BufferedInputStream(
                new FileInputStream(filename)))) {
        graph = new Graph();
        Map<Integer, Node> idNodeMap = new HashMap<>();
        int nodeCount = in.readInt();
        for (int i = 0; i < nodeCount; i++) {
            int id = in.readInt();
            Node node = new Node();
            idNodeMap.put(id, node);
            graph.getNodes().add(node);
        }
        for (Node node : graph.getNodes()) {
            node.setLabel(in.readUTF());
            int edgeCount = in.readInt();
            for (int i = 0; i < edgeCount; i++) {
                node.getEdges().add(idNodeMap.get(in.readInt()));
    } } }
    return graph;
}
```

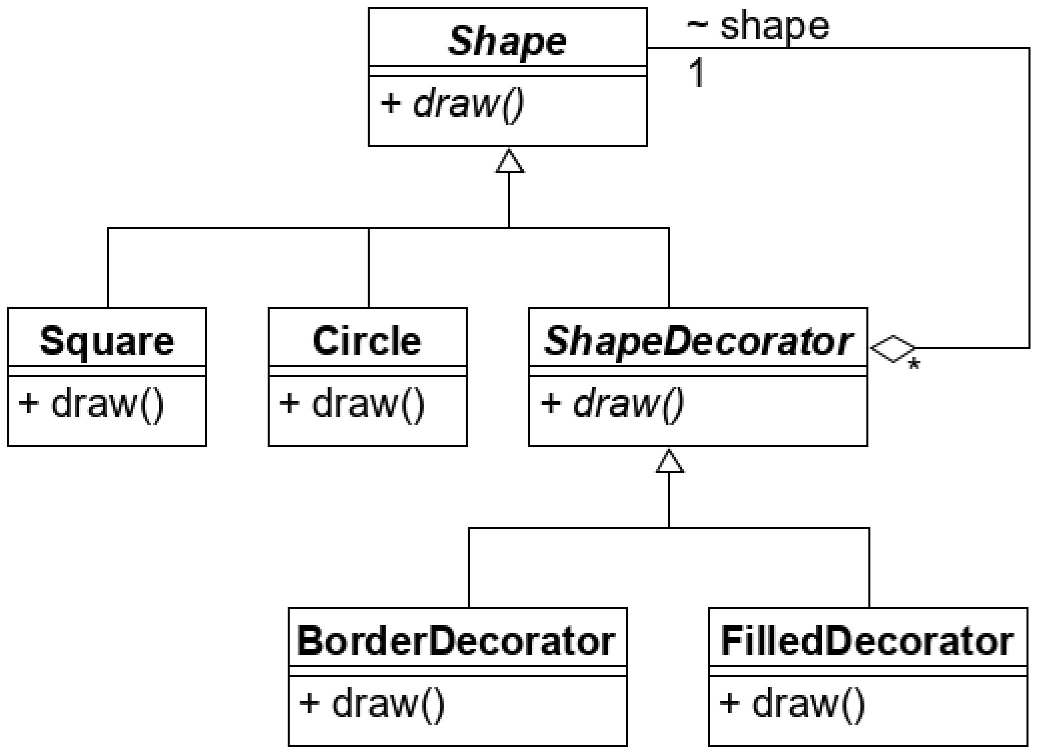
CARL VON OSSIETZKY universität OLDENBURG

## Decorator Pattern

# Decorator Pattern



## Decorator Pattern: Beispiel



# Decorator Pattern: Beispiel

```java
public abstract class Shape {
    public abstract void draw();
}
```

```java
class Square extends Shape {
    @Override
    public void draw() {
        // draw square
    }
}
```

```java
class Circle extends Shape {
    @Override
    public void draw() {
        // draw circle
    }
}
```

```java
abstract class ShapeDecorator extends Shape {

    Shape shape;

    public ShapeDecorator(Shape shape) {
        this.shape = shape;
    }
}
```

# Decorator Pattern: Beispiel

```
class BorderDecorator extends ShapeDecorator {

    public BorderDecorator(Shape shape) {
        super(shape);
    }

    @Override
    public void draw() {
        shape.draw();
        // draw border
    }
}
```

```
class FilledDecorator extends ShapeDecorator {

    public FilledDecorator(Shape shape) {
        super(shape);
    }

    @Override
    public void draw() {
        shape.draw();
        // draw filled shape
    }
}
```

# Decorator Pattern: Beispiel

```java
Shape square = new Square();
square.draw();
// draw square

Shape filledSquare = new FilledDecorator(new Square());
filledSquare.draw();
// draw square
// draw filled shape

Shape filledBorderedCircle = new FilledDecorator(
           new BorderDecorator(new Circle()));
filledBorderedCircle.draw();
// draw circle
// draw border
// draw filled shape
```

## Decorator Pattern: Beispiel (2)

```java
DataOutputStream out =
        new DataOutputStream(
                new BufferedOutputStream(
                        new FileOutputStream(filename)));
```

## Serializable

# Serialisierung

- Java-Standardverfahren zum Speichern und Laden von Objekten
  - auch zur Übertragung im Netzwerk
- Einfach zu implementieren
- Objekt-Identität bleibt erhalten
- Ineffizient bzgl. Laufzeit und Speicherplatz
- Geeignet für kleine Datenmengen

## Dominos speichern (4)

```
public class Domino implements Serializable {

    private static final long serialVersionUID = 1L;

    public static void save(String filename, Domino piece)
                throws IOException {
        try (ObjectOutputStream out = new ObjectOutputStream(
                    new BufferedOutputStream(new FileOutputStream(filename)))) {
            out.writeObject(piece);
        }
    }
}
```

## Dominos laden (4)

```
public class Domino implements Serializable {

    private static final long serialVersionUID = 1L;

    public static Domino load(String filename)
                throws IOException, ClassNotFoundException {
        Domino piece = null;
        try (ObjectInputStream in = new ObjectInputStream(
                new BufferedInputStream(new FileInputStream(filename)))) {
            piece = (Domino) in.readObject();
        }
        return piece;
    }
}
```

## Graphen speichern (2)

```
public class Graph implements Serializable {

    private static final long serialVersionUID = 1L;

    public static void save(String filename, Graph graph) throws IOException {
        try (ObjectOutputStream out = new ObjectOutputStream(
                    new BufferedOutputStream(new FileOutputStream(filename)))) {
            out.writeObject(graph);
        }
    }
}
```

## Graphen laden (2)

```
    public static Graph load(String filename)
                throws IOException, ClassNotFoundException {
        Graph graph = null;
        try (ObjectInputStream in = new ObjectInputStream(
                new BufferedInputStream(new FileInputStream(filename)))) {
            graph = (Graph) in.readObject();
        }
        return graph;
    }
}
```

## Verwendung von Serializable

```
Exception in thread "main" java.io.NotSerializableException: Node
```

```
class Node implements Serializable { … }
```

Bei der Serialisierung müssen **alle** Klassen, die Teil der Struktur sind, serialisierbar sein.

# `transient`

- Standardmäßig serialisiert Java alle nicht-statischen Attribute einer Klasse
- Mit dem Schlüsselwort **`transient`** können einzelne Attribute von der Serialisierung ausgeschlossen werden
- Dies wird insbesondere verwendet für
  - abgeleitete Attribute, d.h. Attribute deren Wert sich aus anderen Attributen ergibt
    - z.B. eine Thumbnail-Vorschau zu einem Bild
  - Attribute, die sich nicht persistieren lassen
    - z.B. Datenbankverbindungen oder Threads

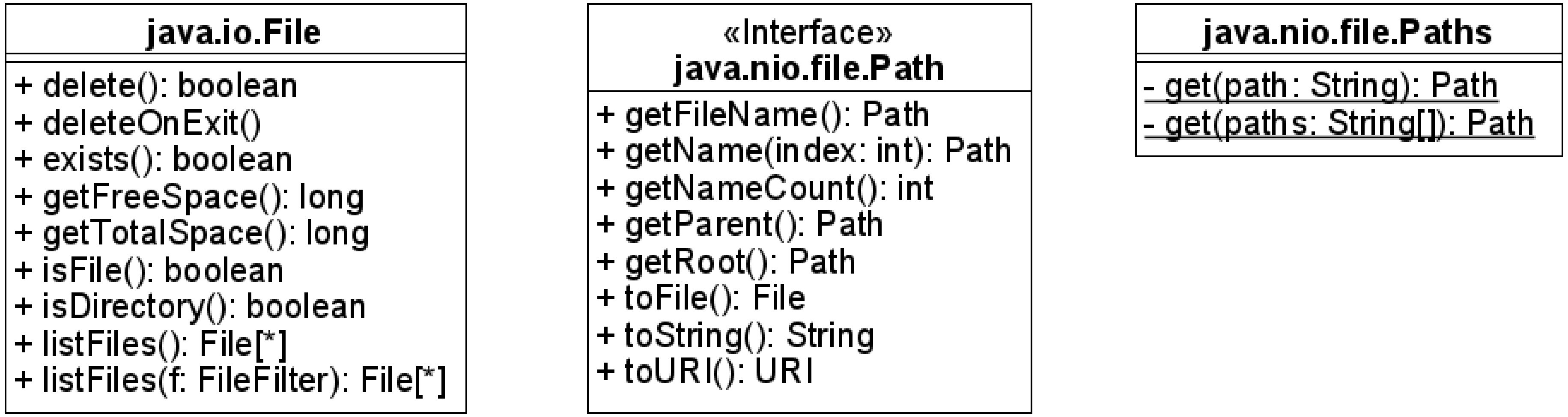
CARL VON OSSIETZKY universität OLDENBURG

## Umgang mit Dateien

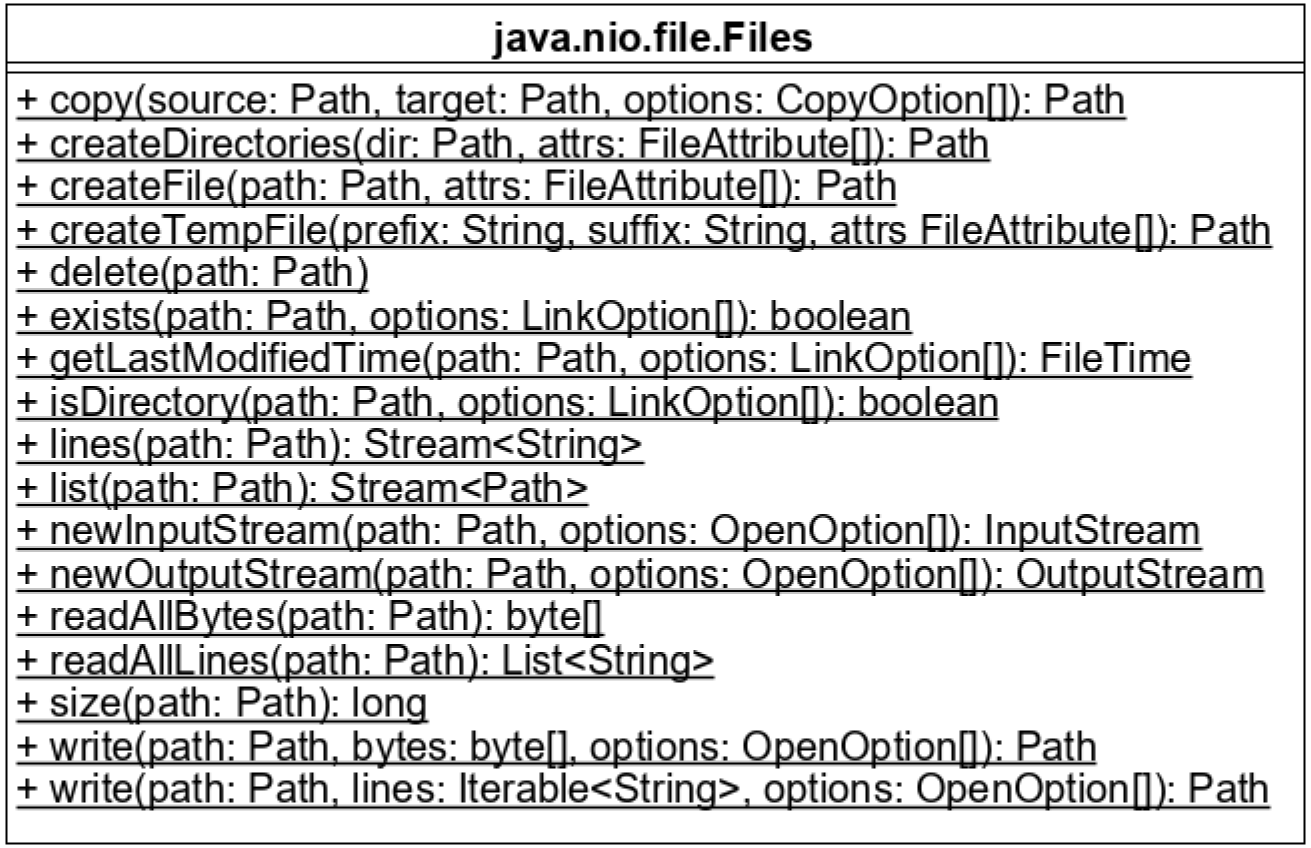
## Dateikopieren mit Java: Beispiel

```java
CopyOption[] options = new CopyOption[] {
        java.nio.file.StandardCopyOption.REPLACE_EXISTING,
        java.nio.file.StandardCopyOption.COPY_ATTRIBUTES,
        java.nio.file.LinkOption.NOFOLLOW_LINKS
};

Path source = Paths.get("path/to/file/filename");
Path dest = Paths.get("other/path/filename2");

try {
    Files.copy(source, dest, options);
} catch (IOException e) {
    e.printStackTrace();
}
```

# Interface **Path**



# Klasse **Files**



**java.nio.file.Files**

+ copy(source: Path, target: Path, options: CopyOption[]): Path
+ createDirectories(dir: Path, attrs: FileAttribute[]): Path
+ createFile(path: Path, attrs: FileAttribute[]): Path
+ createTempFile(prefix: String, suffix: String, attrs FileAttribute[]): Path
+ delete(path: Path)
+ exists(path: Path, options: LinkOption[]): boolean
+ getLastModifiedTime(path: Path, options: LinkOption[]): FileTime
+ isDirectory(path: Path, options: LinkOption[]): boolean
+ lines(path: Path): Stream<String>
+ list(path: Path): Stream<Path>
+ newInputStream(path: Path, options: OpenOption[]): InputStream
+ newOutputStream(path: Path, options: OpenOption[]): OutputStream
+ readAllBytes(path: Path): byte[]
+ readAllLines(path: Path): List<String>
+ size(path: Path): long
+ write(path: Path, bytes: byte[], options: OpenOption[]): Path
+ write(path: Path, lines: Iterable<String>, options: OpenOption[]): Path

## Dateioperationen: Beispiel

```
Path file = Paths.get("file.txt");
if (!Files.exists(file)) {
    Files.createFile(file);
}
List<String> lines = Files.readAllLines(file);
lines.add(Long.toString(Files.size(file)));
Files.write(file, lines);
```

```
0
```

```
0
3
```

```
0
3
6
```

## Lernziele

- **`InputStream`**, **`OutputStream`**
- Decorator-Pattern
- **`Serializable`**
- **`Path`**, **`File`**