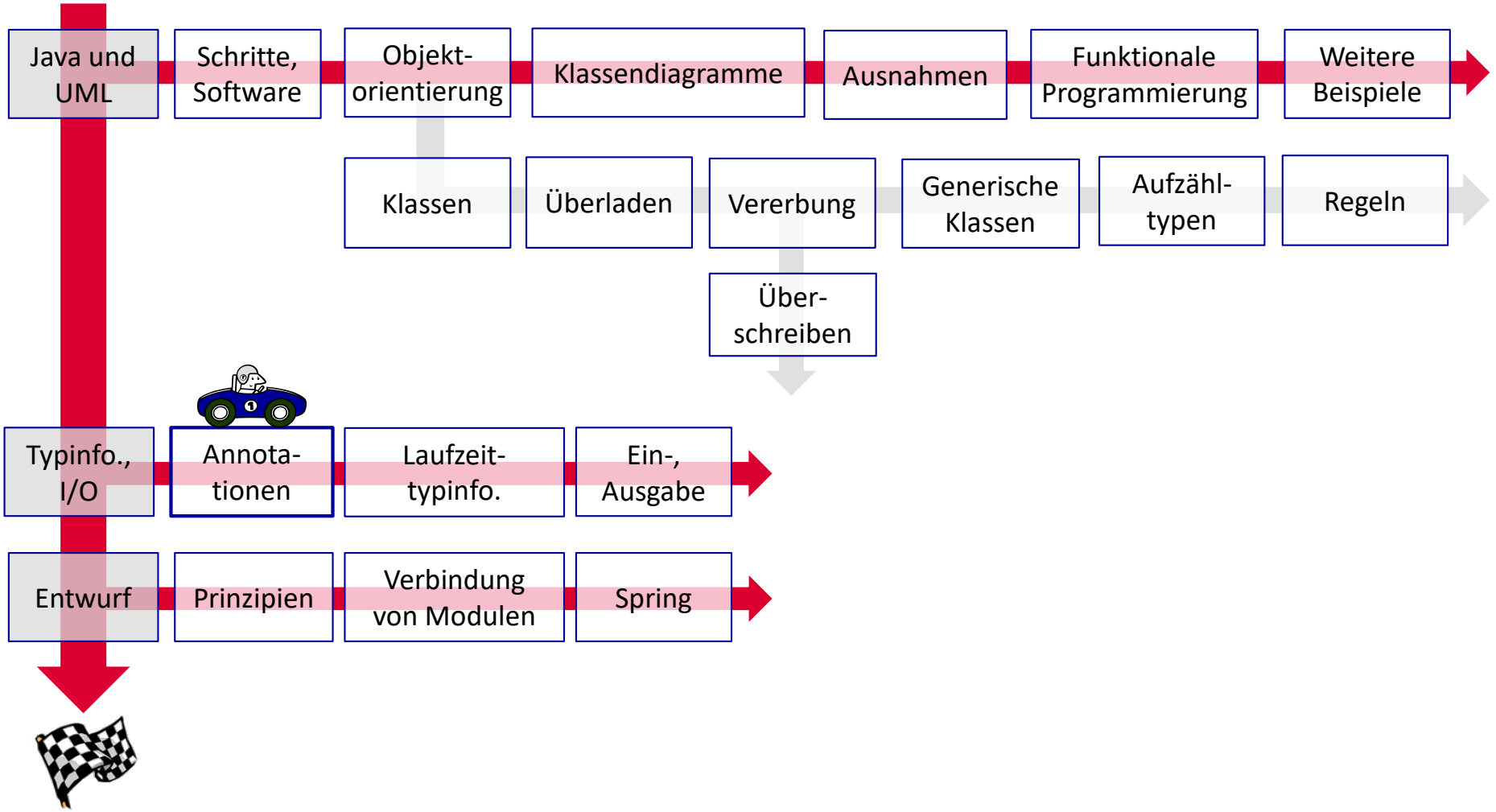
# Informatik 2
## Typinformationen und Ein- und Ausgabe

Prof. Dr.-Ing. Holger Vogelsang
holger.vogelsang@hs-karlsruhe.de

# Inhaltsverzeichnis

# Annotationen
## Übersicht



# Annotationen
## Motivation

- *Wie lassen sich dem Compiler, der virtuellen Maschine oder dem laufenden Programm Zusatzinformationen übergeben?*
- *Wie können eigene Zusatzinformationen erstellt werden?*

### Annotationen als Metainformationen im Programm

- Annotationen sind Zusatzinformationen, die im Programmtext abgelegt werden, damit sie „jemand“ später auswerten kann.
- Beispiele:
  - `@Override`:
    - Gibt dem Compiler den Hinweis, dass diese Methode eine Methode der Basisklasse überschreibt.
    - Der Compiler kann prüfen, ob das wirklich der Fall ist.
  - `@Deprecated`:
    - Gibt dem Compiler den Hinweis, dass das markierte Element (Methode, Klasse, …) veraltet ist und nicht mehr verwendet werden sollte.
    - Der Compiler gibt eine Warnung aus, wenn jemand das Element trotzdem nutzt.

- **`@SuppressWarnings`**:
  - Unterdrückt bestimmte oder alle Compiler-Warnungen:

    **`ArrayList al = new ArrayList();`**

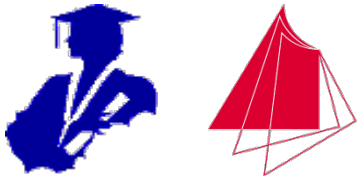
    Ergibt die Compiler-Warnung, dass die generische Klasse ohne Typparameter verwendet wurde („raw type“).

    Die Warnung kann mit

    **`@SuppressWarnings("rawtypes")`** abgeschaltet werden:

```
@SuppressWarnings("rawtypes")
ArrayList al = new ArrayList();
```

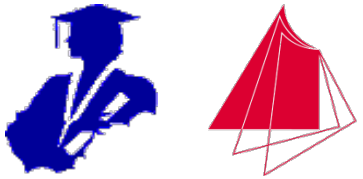
# Annotationen
## Metainformationen



- Annotationen können also zusätzliche Informationen übergeben bekommen:
  - **`@Annotationstyp`**: Marker-Annotation (keine Parameter)
  - **`@Annotationstyp(wert)`**: Annotation mit genau einem Wert (s.o.)
  - **`@Annotationstyp(schlüssel1=wert1,schlüssel2=wert2,…)`**: Annotation mit Schlüssel/Werte-Paaren
- Hinweise zur Syntax:
  - **`@Annotationstyp("wert")`** ist eine Kurzschreibweise für **`@Annotationstyp({"wert"})`** und das ist eine Kurzschreibweise für **`@Annotationstyp(value={"wert"})`**
- Auswertung der Annotationen:
  - Source: Beispiele siehe oben, die Annotationen werden vom Compiler ausgewertet und entfernt.
  - Class: Die Annotationen werden in den Bytecode geschrieben und von der virtuellen Maschine ausgewertet und entfernt.
  - Runtime: Die Annotationen werden in den Bytecode geschrieben und vom Programm ausgewertet (Beispiele folgen).

# Annotationen

## Metainformationen: Definition



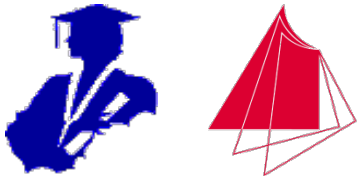
- Definition von Annotationen (nicht vollständig)
- Beispiel **`@Override`**:

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

- Hinweise:
  - **`@interface`**: Definition einer Annotation
  - **`@Target`** ist eine weitere Annotation, die angibt, wo die neu definierte Annotation verwendet werden darf.
    - **`ElementType`** ist ein Aufzähltyp. **`@Override`** kann also nur bei Methoden verwendet werden.
    - **`@Target`** erlaubt die Angabe mehrerer Ziel-Typen.
  - **`@Retention`** ist eine Annotation, die angibt, wann die neue Annotation ausgewertet wird (ihre Lebensdauer):
    - **`RetentionPolicy`** ist auch eine Aufzählung, die die Werte **`SOURCE`**, **`CLASS`** und **`RUNTIME`** annehmen kann.

# Annotationen

## Metainformationen: Definition



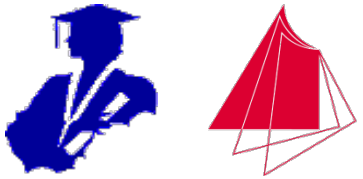
- Definition von Annotationen mit Parametern (nicht vollständig)
- Beispiel `@SuppressWarnings({"rawtypes", "unchecked"})`:

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
  String[] value();
}
```

- Die Parameter werden in die einzelnen Zellen des String-Arrays geschrieben.

# Annotationen

## Metainformationen: Definition



- Definition von Annotationen mit Schlüssel/Werte-Paaren (nicht vollständig)
- Beispiel anhand der fiktiven Runtime-Annotation **`@Author({first="Holger", last="Vogelsang"})`**:

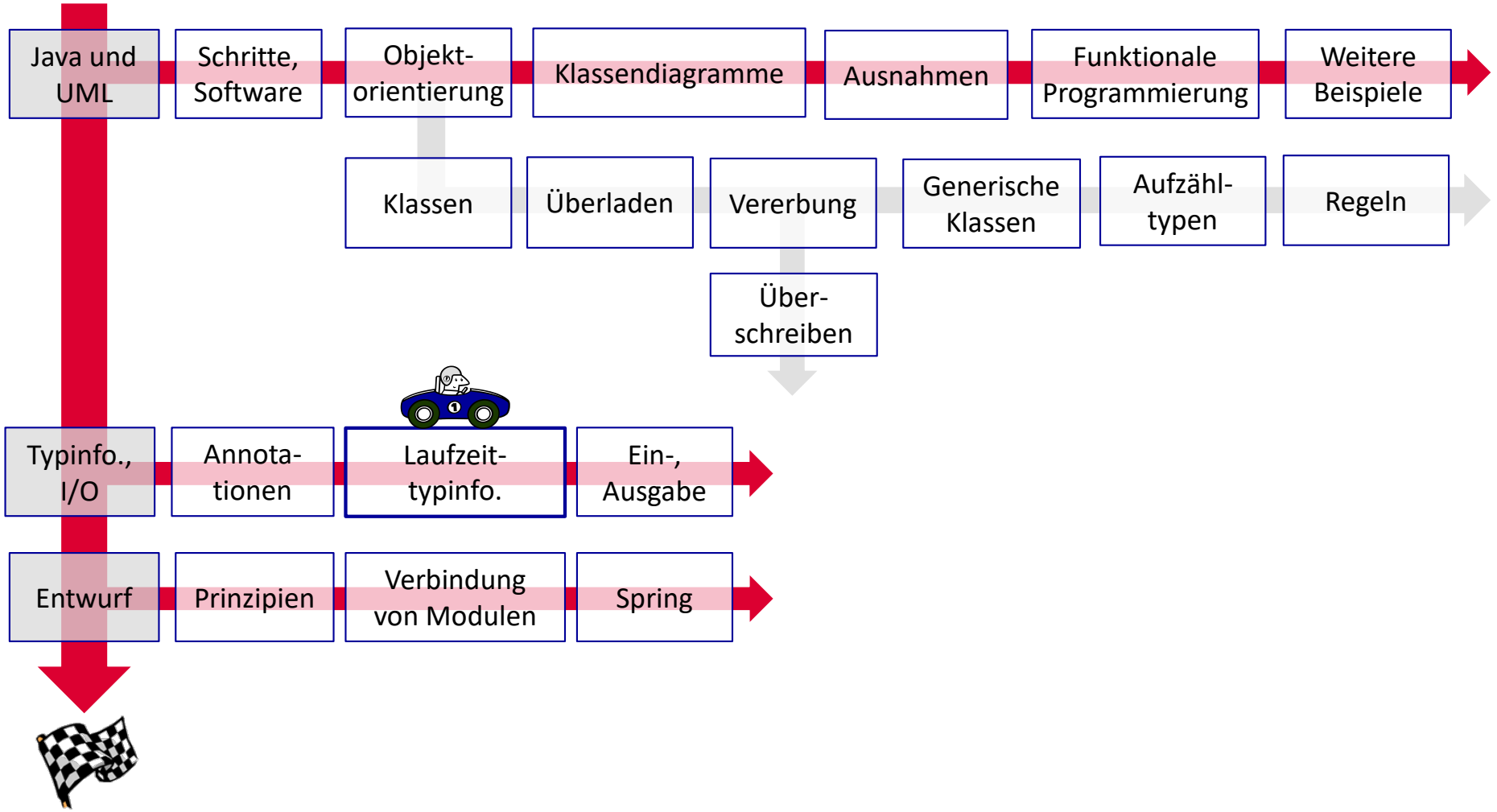
```
@Target({TYPE, FIELD, METHOD, CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
  String first();
  String last();
  String lastChanged() default "09.02.2012";
  boolean isFinished() default false;
}
```

- Die Schlüssel-Namen entsprechen den Methoden-Namen.
- Werte, die mit **`default`** markiert sind, erhalten den Standardwert, wenn keiner angegeben wurde.

- Was sollen Sie mit diesen Informationen anfangen???
  - Kommt gleich bei den Laufzeit-Typinformationen („reflection“)!

# Laufzeit-Typinformationen

## Übersicht



# Laufzeit-Typinformationen

## Motivation

- *Wie können Informationen über Klassen, Schnittstellen usw. zur Laufzeit ermittelt werden?*
- *Wozu ist das sinnvoll?*

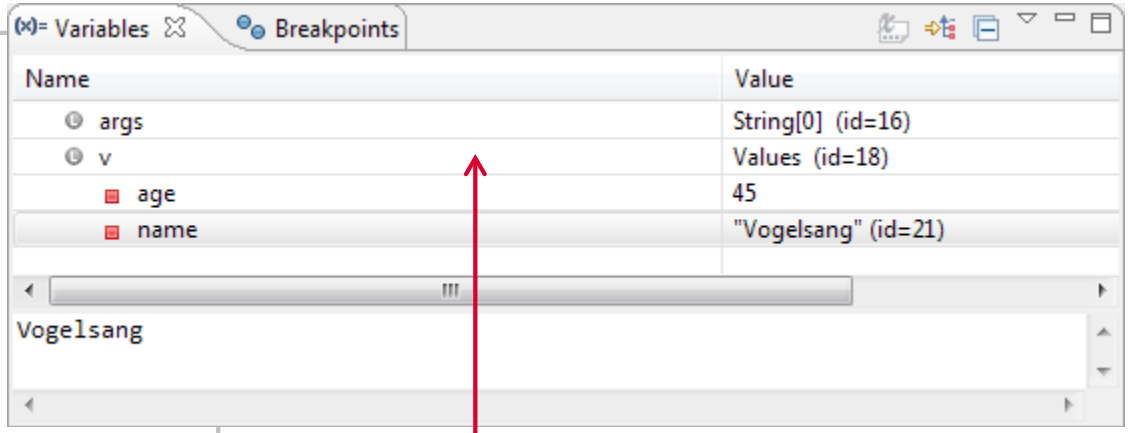
# Laufzeit-Typinformationen
## Einführung

- Warum sollte man zur Laufzeit Informationen zu Klassen, Schnittstellen usw. benötigen?
  - Frameworks könnten Objekte aus unbekannten Klassen zur Laufzeit erzeugen. Ein Beispiel zur Modularisierung mit Spring kommt später noch.
  - Debugger benötigen Zugriff auf private Attribute eines Objektes.
  - Klassen-Browser und Entwicklungsumgebungen:
    - Klassenbrowser müssen den Aufbau einer Klasse kennen, um z.B. dem Nutzer interaktiv die Veränderung eines Wertes zu erlauben.
    - Eine IDE muss die Klassen kennen, um kontextsensitive Hilfe anbieten zu können.
    - GUI-Builder müssen die Oberflächen-Klassen untersuchen können, um passende Editoren dynamisch erzeugen zu können.
    - Objekte müssen auf Festplatte gespeichert oder über das Netzwerk übertragen werden.
- Laufzeit-Typinformationen sind Metadaten = Daten über Daten.
- Die folgende Einführung ist unvollständig.

# Laufzeit-Typinformationen
## Einführung

- Beispiel zum Debugger:

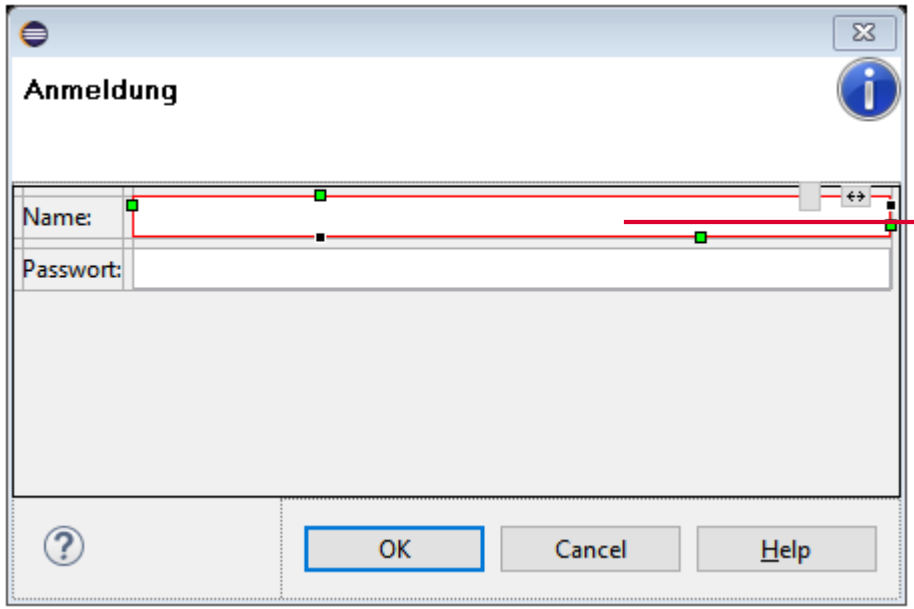
```
class Values {
  private String name;
  private int    age;
  public Values(String name,
                int age) {
    this.name = name;
    this.age = age;
  }
}

public class Test {
  public static void main(String[] args) {
    Values v = new Values("Vogelsang", 45);
    System.out.println(v);
  }
}
```



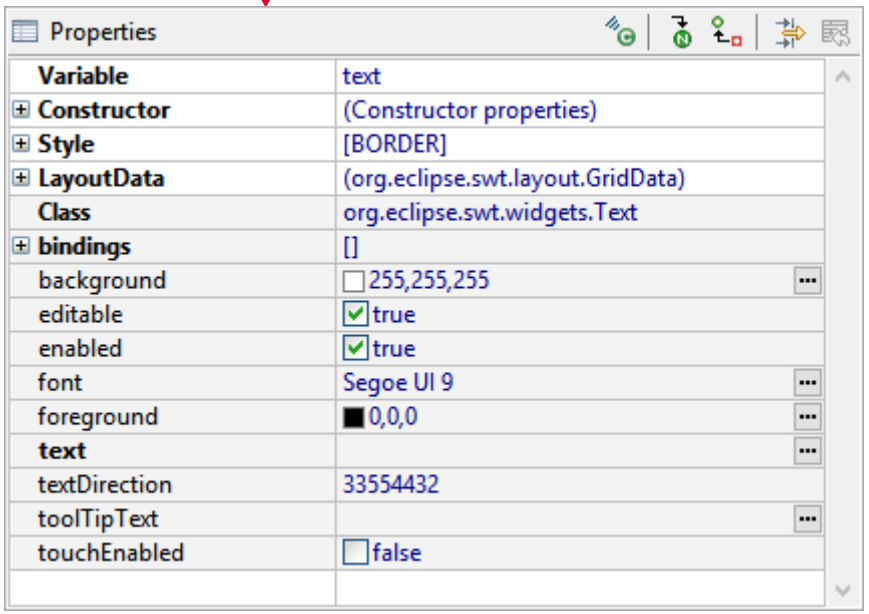

Stopp auf Breakpoint:
Ansicht der privaten Daten möglich

# Laufzeit-Typinformationen
## Einführung

- Beispiel zum GUI-Builder:




erzeugt dynamisch einen
Editor, passend zu den
(auch privaten) Daten der Klasse



# Laufzeit-Typinformationen
## Einführung

- Nachteile von Reflection:
  - Die Zugriffe kosten relativ viel Zeit.
  - Reflection funktioniert nicht, wenn ein Security-Manager (Applets) das verhindert.
  - Es wird ein Zugriff auf interne Attribute von Objekten ermöglicht:
    - Einerseits lässt sich damit die Kapselung aushebeln.
    - Andererseits kann es sinnvoll sein, bei unterschiedlichen Aufgaben den Zugriff auf private Daten zu erlauben (z.B. bei Speicherung).
- Wie erfolgt der Zugriff auf die Typinformationen?
  - In der Basisklasse **`Object`** gibt es die Methode **`Class<?> getClass()`**. Sie liefert ein Klassenobjekt mit Informationen zur Klasse des Objektes zurück.

    Beispiel: **`Class<?> myClass = xyzObject.getClass();`**
  - Bei primitiven Datentypen gibt es das Attribut **`class`**.

    Beispiel: **`Class<?> intClass = int.class;`**
  - Bei Klassen und Schnittstellen gibt es ebenfalls das Attribut **`class`**.

    Beispiel: **`Class<?> integerClass = Integer.class;`**

# Laufzeit-Typinformationen
## Einführung

- Klassen lassen sich auch dynamisch anhand ihres Namens laden. Beispiel:

```
Class<?> myClass = Class.forName("de.hska.iwii.i2.Example");
```

- Das ist sinnvoll für Klassen, deren Namen zur Übersetzungszeit noch nicht feststanden (z.B. über das Netzwerk geladen). Achtung: Es können Ausnahmen auftreten, wenn z.B. die Klasse gar nicht existiert.

- Es gibt immer nur jeweils ein `Class`-Objekt je Klasse oder primitiven Datentyp.
- Was beschreibt das `Class`-Objekt u.A.?
  - Handelt es sich um ein Array, einen primitiven Datentyp, einen Aufzähltyp oder um eine Klasse?
  - Welche Attribute oder Aufzählwerte sind vorhanden?
  - Welche Methoden und Konstruktoren gibt es?
  - Welche Annotationen sind vorhanden?
  - Von welcher Basisklasse erbt die Klasse und welche Schnittstellen implementiert sie?
  - Wie sehen die Zugriffsrechte aus?

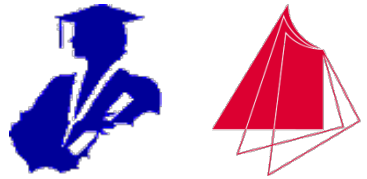
# Laufzeit-Typinformationen
## Einführung

- Was lässt sich mit dem **`Class`**-Objekt alles machen (Auswahl)?
  - Vergleich, ob zwei Objekte exakt derselben Klasse entstammen. Beispiel (wenn die Klassen der Objekte vom selben Classloader geladen wurden):

    ```
    if (objRef1.getClass() == objRef2.getClass())
    ```
  - neue Objekte erzeugen
  - lesend und schreiben auf private Attribute und Methoden zugreifen:
    - Daten auslesen und auf Festplatte schreiben → Serialisierung (kommt im Kapitel zu Ein- und Ausgabe)
    - Objekte anhand gespeicherter Daten wiederherstellen → auch Serialisierung
    - Objektinspektor bauen
    - auf Basis der Annotationen Zusatzfunktionalität anbieten (automatische Ablage in der Datenbank, Überprüfung der Attributwerte, …) → Beispiel kommt gleich
    - …

# Laufzeit-Typinformationen

## Einführung: Beispielklasse

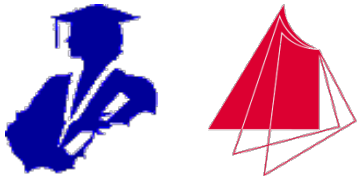


- Die folgenden Beispiele agieren auf der rudimentären Klasse **`Fraction`** für Brüche:

```
public class Fraction {
  private long counter;
  private long denominator;

  public Fraction(long counter, long denominator) {
    this.counter = counter;
    this.denominator = denominator;
  }
  public Fraction() {
    this(0, 1);
  }

  public void setCounter(long counter) {
    this.counter = counter;
  }

  public long getCounter() {
    return this.counter;
  }
}
```

# Laufzeit-Typinformationen

## Einführung: Konstruktoraufrufe



- Mit Hilfe des `Class`-Objektes lassen sich Objekte erzeugen:
  - Standardkonstruktor, Variante 1:

```
Object newObject = Class.forName("de.hska.iwii.i2.Fraction").newInstance();
```

  - Variante 2:

```
Class<?> classForName = Class.forName("de.hska.iwii.i2.Fraction");
Constructor<?> constructor = classForName.getConstructor();
Object newObject = constructor.newInstance();
```

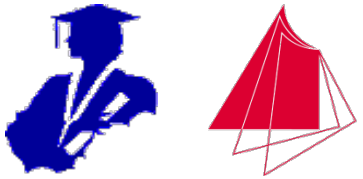
  - Es lassen sich auch beliebige Konstruktoren aufrufen:

```
Class<?> classForName = Class.forName("de.hska.iwii.i2.Fraction");
// Konstruktor mit zwei long-Parametern
Constructor<?> constructor = classForName.getConstructor(long.class,
                                                         long.class);
Object newObject = constructor.newInstance(42L, 66L);
```

- So lassen sich Objekte aus Klassen erzeugen, die erst zur Laufzeit bekannt wurden.

# Laufzeit-Typinformationen

## Einführung: Attributinformationen



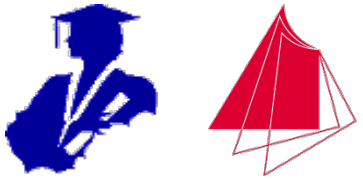
- Ausgabe der Informationen zu allen Attributen einer Klasse:

```
public void printAttributes(String className) throws ClassNotFoundException {
  System.out.println("Attribute der Klasse " + className);

  // Referenz auf die Klasse holen
  Class<?> classForName = Class.forName(className);

  // Array aller Attributinformationen holen
  Field[] attributes = classForName.getDeclaredFields();

  // einzelne Attributinformationen ausgeben
  for (Field attr: attributes) {
    System.out.println("  " + attr.getName() + ": " + attr.getType()
                       + " (" + attr.getModifiers() + ")");
  }
}
```

- Methoden der Klasse **`Field`**:
  - **`String getName()`**: liefert den Attributnamen
  - **`Class<?> getType()`**: liefert die Klasse des Attributs
  - **`int getModifiers()`**: liefert eine Bitkombination der Modifizierer wie **`public`**, …

# Laufzeit-Typinformationen

## Einführung: Attribute auslesen



- Mit Hilfe der Reflection-API können Attribute eines unbekannten Objektes ausgelesen werden (Ausnahmen sind weggelassen):

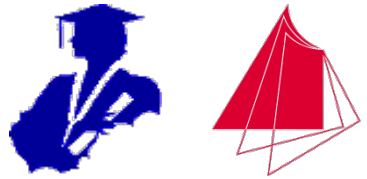
```
// Liest das Attribut mit dem Namen attributeName aus dem Objekt object.
public Object readAttribute(Object object, String attributeName) {
  // Referenz auf das Klassenobjekt holen
  Class<?> classRef = object.getClass();

  // Referenz auf das gesuchte Attribut holen
  Field field = classRef.getDeclaredField(attributeName);

  // Zugriff ermöglichen, auch wenn es eventuell privat ist
  field.setAccessible(true);

  // Wert des Attributes auslesen
  return field.get(object);
}
```

- Aufruf mit:

```
Object newObject = Class.forName("de.hska.iwii.i2.Fraction").newInstance();
Object value = readAttribute(newObject, "counter");
// Value ist ein Long-Objekt mit dem Wert 0.
```

# Laufzeit-Typinformationen

## Einführung: Attribute beschreiben



- Mit Hilfe der Reflection-API können Attribute eines unbekannten Objektes beschrieben werden (Ausnahmen sind wieder weggelassen):

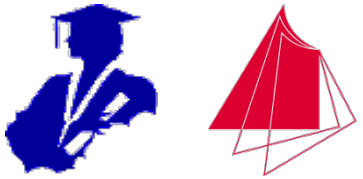
```
// Schreibt den Wert value in das Attribut attributeName des Objekts object
public void writeAttribute(Object object, String attributeName, Object value) {
  // Referenz auf das Klassenobjekt holen
  Class<?> classRef = object.getClass();

  // Referenz auf das gesuchte Attribut holen
  Field field = classRef.getDeclaredField(attributeName);

  // Zugriff ermöglichen, auch wenn es eventuell privat ist
  field.setAccessible(true);

  // Wert des Attributes ändern
  field.set(object, value);
}
```

- Aufruf mit:

```
Object newObject = Class.forName("de.hska.iwii.i2.Fraction").newInstance();
writeAttribute(newObject, "counter", 9999L);
```

# Laufzeit-Typinformationen

## Einführung: Annotationen auslesen



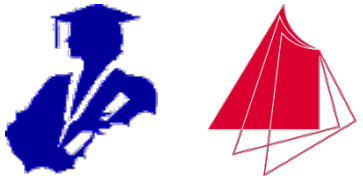
- Runtime-Annotationen können aus dem **`Class`**-Objekt ausgelesen werden.
- Erweiterung des **`Fraction`**-Beispiels um eine eigene Annotation:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Property {
  String name() default "";
}

public class Fraction {
  @Property(name="zaehler")
  private long counter;

  private long denominator;
  // Rest unverändert
}
```

- Mit der fiktiven Annotation **`Property`** wird ein optionaler Name vergeben werden, den das Attribut in der Datenbank erhalten soll.

# Laufzeit-Typinformationen

## Einführung: Annotationen auslesen



- Zugriff auf die Annotation **Property**:

```
public String getPropertyAnnotation(String className, String attributeName) {
  Class<?> classForName = Class.forName(className);
  Field attribute = classForName.getDeclaredField(attributeName);
  if (attribute != null) {
    Annotation annon = attribute.getAnnotation(Property.class);
    if (annon != null) {
      Property prop = (Property) annon;
      return prop.name();
    }
  }
  return null;
}
```

"counter"

```
public class Fraction {

  @Property(name="zaehler")
  private long counter;

  private long denominator;
  // Rest unverändert
}
```

- So kann die Anwendung Annotationen auslesen und Zusatzfunktionalität bereitstellen. Es sinnvolleres Beispiel ist im Kapitel zur Modularisierung mit Spring vorhanden.

# Laufzeit-Typinformationen

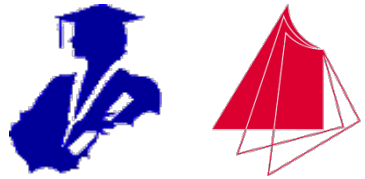
## Demo

- Im Paket **`javax.lang.model`** und seinen Unterpaketen befinden sich spezielle Klassen, um Annotationen auszulesen → soll hier nicht behandelt werden.

- Kleine Demo am Laptop:
  - Reflection
  - Dynamic Proxy

# Laufzeit-Typinformationen

## Fazit



- Was kann Reflection denn nun in der Praxis?
  - Durch Reflection lassen sich alle Daten aus Objekten auslesen und wieder einschreiben. Für das Konvertieren in XML gibt es:
    - **`XMLEncoder`**: Objekte in ein XML-Format wandeln, um sie auf einen Stream (kommt gleich) schreiben zu können.
    - **`XMLDecoder`**: Aus XML-Daten werden wieder Objekte.
  - Es können unbekannte Klassen zur Laufzeit geladen und untersucht werden. Das ist wichtig für Werkzeuge wie Debugger und GUI-Editoren.

- Warnung: Reflection ist ein Mittel zum Bau von Werkzeugen oder sogenannten Frameworks. Es sollte in „normalen“ Anwendungen möglichst gemieden werden.

# Laufzeit-Typinformationen und Debugging

## Untersuchung eines Programms zur Laufzeit

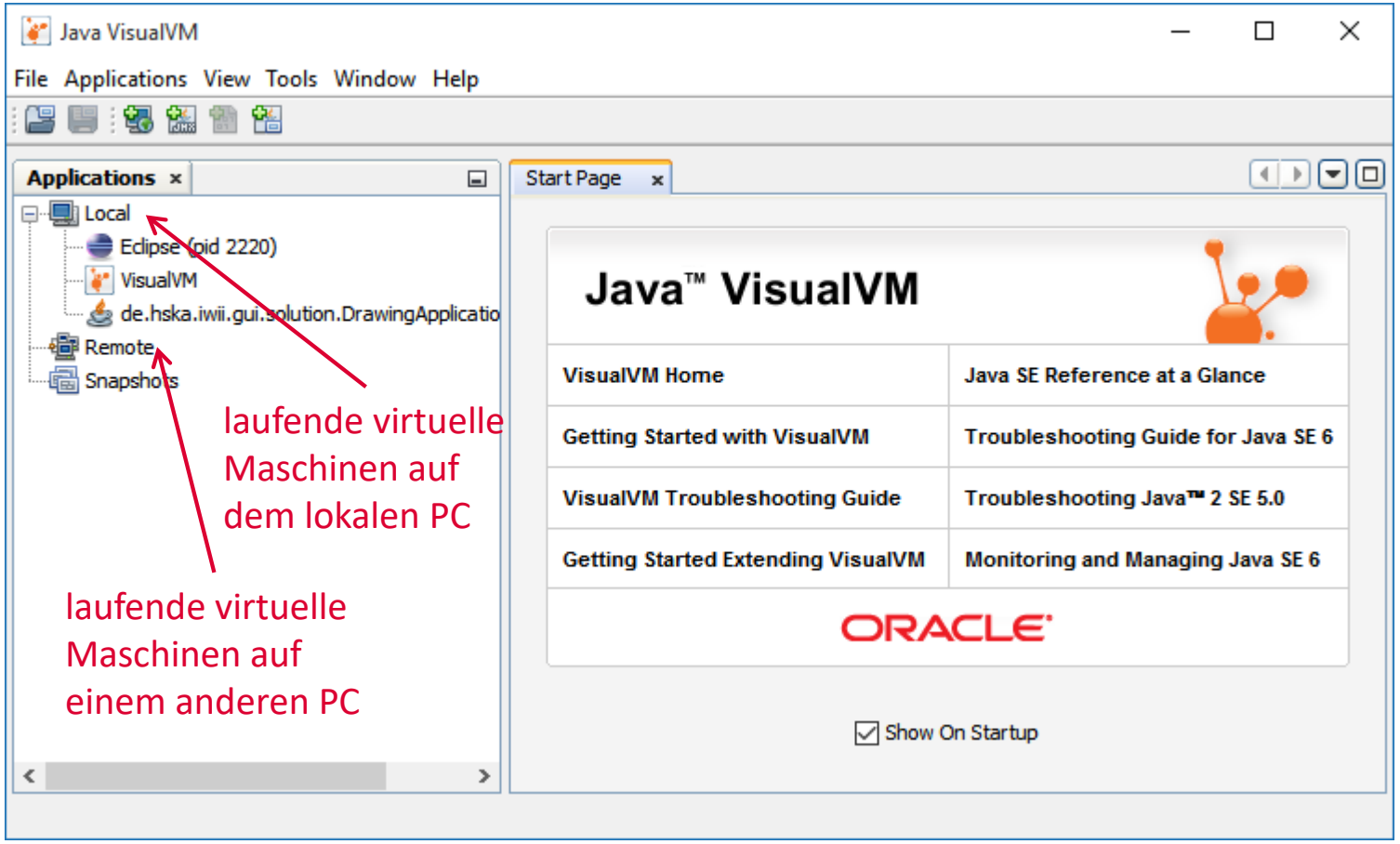
- Hilfe!!
  - Was macht mein Programm eigentlich während der Laufzeit?
  - Wieso ist immer weniger freier Heap-Speicher vorhanden?
- Zur Untersuchung eines laufenden Programms hilft ein Profiler: Laufzeitverhalten, Speicherverbrauch, …
- Java liefert bereits einen einfachen Profiler mit: Java Visual VM, Datei unter Windows

  `<jdk-verzeichnis>/bin/jvisualvm.exe`

- Dieses Kapitel gehört nicht direkt zu den Laufzeit-Typinformationen.

# Laufzeit-Typinformationen und Debugging

## Untersuchung eines Programms zur Laufzeit



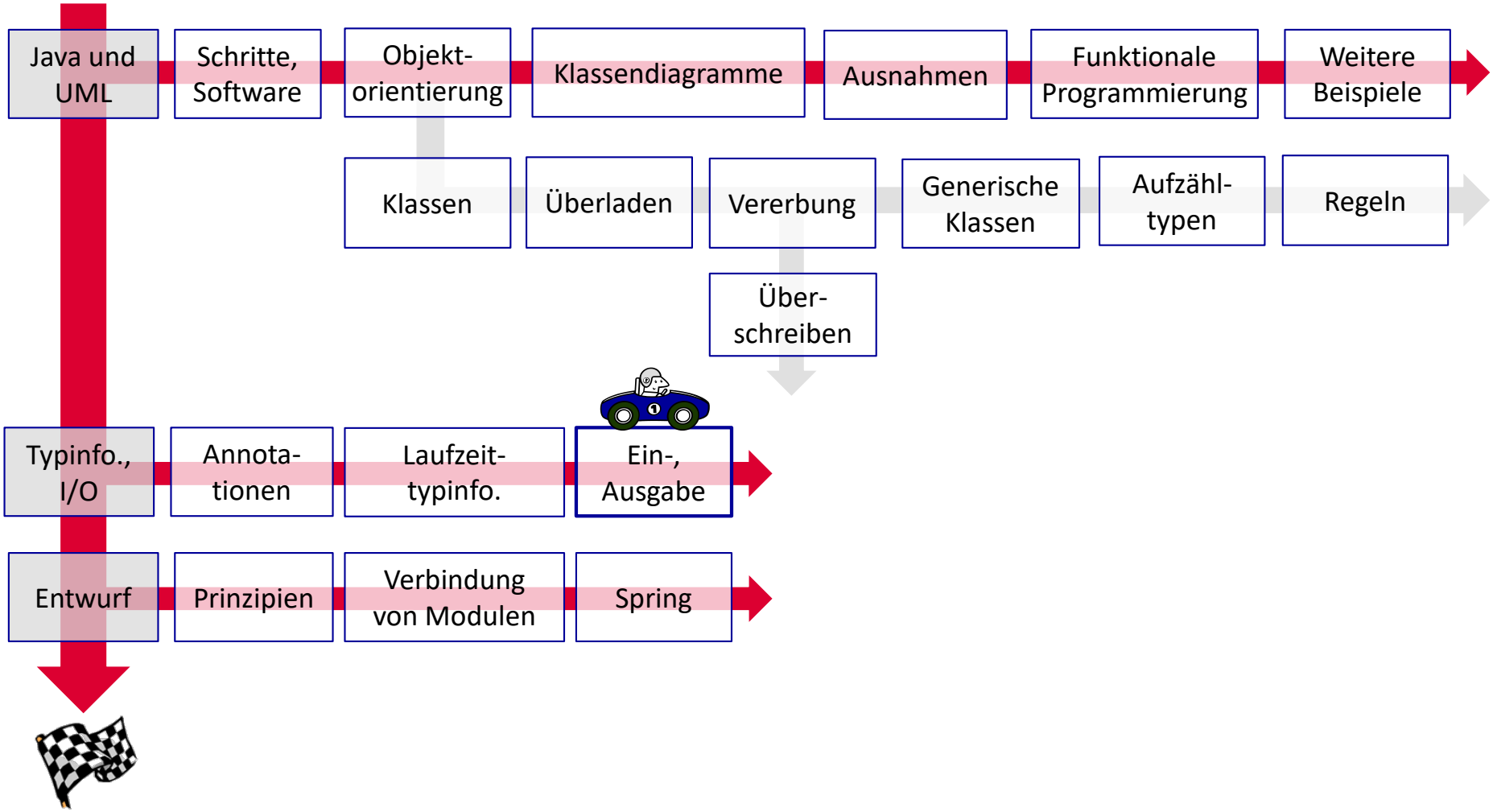
# Laufzeit-Typinformationen und Debugging

## Untersuchung eines Programms zur Laufzeit

- Life-Demo
  - Speicherverbrauch
  - Snapshots
  - Filtern der Anzeige

# Laufzeit-Typinformationen

## Übersicht



# Ein- und Ausgabe
## Motivation

- *Wie werden Daten aus verschiedenen Quellen eingelesen und auf verschiedene Ziele geschrieben?*
- *Wie können die Daten dabei manipuliert werden?*

# Ein- und Ausgabe
## Idee

### Einsatz

- Programme müssen häufig Daten von einer externen Quelle lesen oder auf ein externes Ziel schreiben.
- Die Information kann überall vorkommen: In einer Datei, auf Festplatte, irgendwo im Internet, im Speicher oder in einem anderen Programm.
- Die Information kann von einem beliebigen Typ sein: Objekte, Zeichen, Bilder, Sounds, Videos,...
- Java besitzt für einen einheitlichen Zugriff auf solche Informationen die Stream-Klassen des Pakets **`java.io`**.
- Interessant ist auch der Inhalt aus **`java.nio.file`** (Kopieren, Umbenennen von Dateien, Dateiattribute, dateisystemspezifische Operationen, …)

### Quelle

- Die folgenden Seiten enthalten eine erweitere Zusammenfassung aus dem Java-Tutorial von Oracle (http://docs.oracle.com/javase/tutorial/).

### Eingabe

- Um Informationen zu lesen, öffnet ein Programm einen Stream auf einer Informationsquelle und liest die Daten sequentiell.


### Ausgabe

- Um Informationen zu schreiben, öffnet ein Programm einen Stream zu einem Informationsziel und schreibt die Daten sequentiell.


Diese Streams haben nicht direkt etwas mit den Streams aus dem Datenstrukturkapitel zu tun.

### Algorithmus

- Unabhängig davon, ob gelesen oder geschrieben wird, ist der Algorithmus fast identisch.

Lesen:

```
open a stream
while has more information {
  read information
}
close the stream
```

Schreiben:

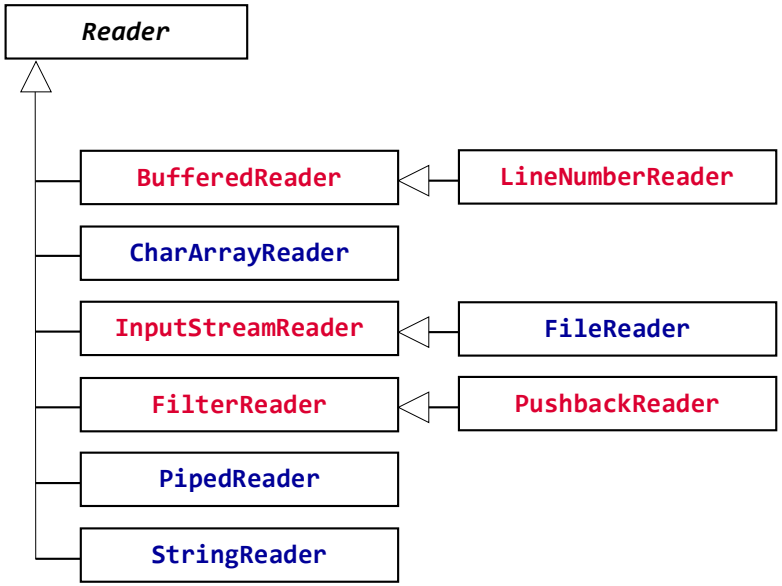
```
open a stream
while has more information {
  write information
}
close the stream
```

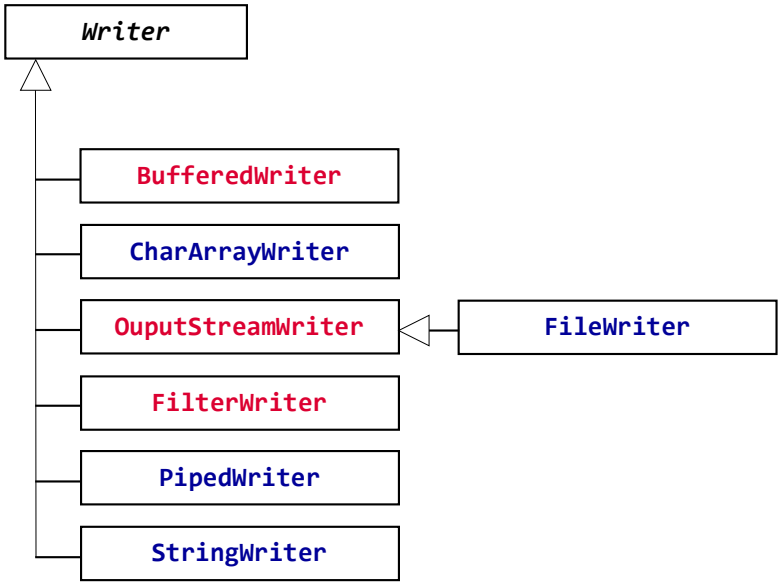
- Die Stream-Klassen bestehen aus zwei Klassen-Hierarchien, basierend auf den zu verarbeitenden Datentypen (Zeichen oder Byte).

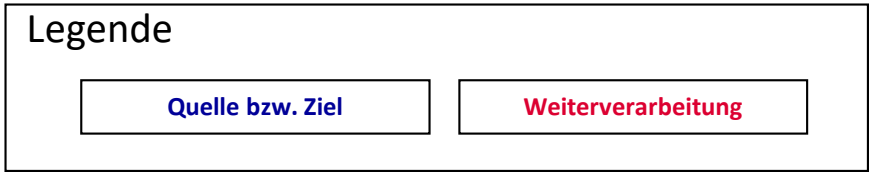
## Zeichen (Character-Streams)

- `Reader` und `Writer` sind die abstrakten Basisklassen für Zeichen-Ströme.
- `Reader` und `Writer` handhaben 16-Bit Unicode-Zeichen.
- Abgeleitete Klassen von `Reader` und `Writer` implementieren spezialisierte Ströme, die sich in zwei Klassen einteilen lassen:
  - Lesen von Datenquellen und Schreiben auf Datenzielen.
  - Bearbeitung von Daten.
- `Reader` und `Writer` werden in der Regel verwendet, um textuelle Informationen zu verwalten, da sie Unicodes handhaben können.
- Byte-Ströme können nur 8-Bit Elemente des Zeichensatzes ISO-Latin-1 erkennen.

- Reader- und Writer Klassen



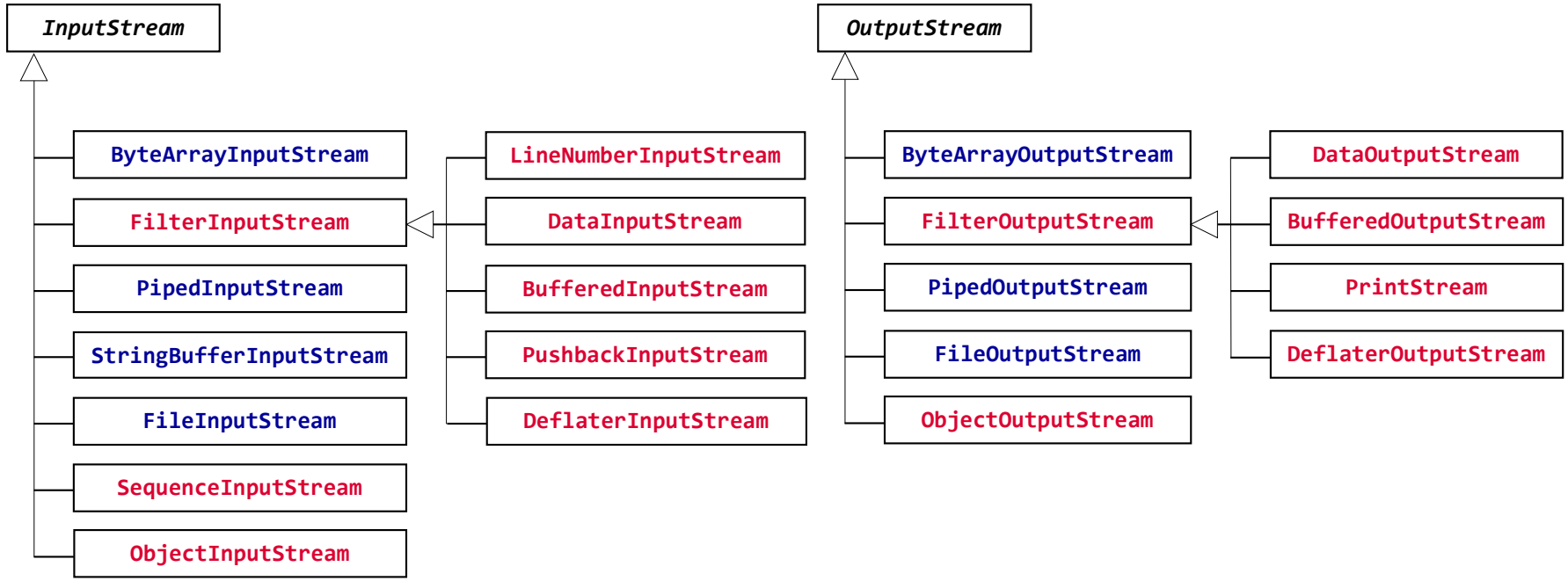



## Bytes (Byte-Streams)

- **`InputStream`** und **`OutputStream`** sind die abstrakten Basisklassen für Byte-Ströme.
- Byte-Ströme werden verwendet, um 8-Bit-Daten zu lesen oder zu schreiben.
- Dazu gehören auch Binärdaten wie Bilder und Sounds.
- Zwei der von **`InputStream`** und **`OutputStream`** abgeleiteten Klassen (**`ObjectInputStream`** und **`ObjectOutputStream`**) werden verwendet, um ganze Objekte zu schreiben oder zu lesen (Serialisierung).
- Die von **`InputStream`** und **`OutputStream`** abgeleiteten Klassen implementieren spezialisierte Ströme, die sich in zwei Klassen einteilen lassen:
  - Lesen von Datenquellen und Schreiben auf Datenziele
  - Bearbeitung von Daten

- Input- und Outputstream-Klassen:



- **`Reader`** und **`InputStream`** definieren ähnliche Signaturen mit unterschiedlichen Datentypen.
- **`Reader`**:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```

- **`InputStream`**:

```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```

- **`Reader`** und **`InputStream`** haben Methoden, um die aktuelle Position im Stream zu markieren, Eingabedaten zu überspringen und die Leseposition zurückzusetzen.

# Ein- und Ausgabe
## Basisklassen zum Schreiben

- Die Basisklassen **`Writer`** und **`OutputStream`** sind sehr ähnlich aufgebaut:
- **`Writer`**:

```
void write(int c )
void write(char cbuf[])
void write(char cbuf[], int offset, int length)
```

- **`OutputStream`**:

```
void write(int c)
void write(byte cbuf[])
void write(byte cbuf[], int offset, int length)
```

- Alle Streamklassen öffnen den zugehörigen Datenstrom automatisch, wenn ein Objekt angelegt wird (bei Aufruf des Konstruktors).
- Ein Datenstrom kann explizit geschlossen werden (Methode **`close`**).
- Der Garbage-Collector schließt einen Strom automatisch, bevor das Objekt gelöscht wird → sollte vorher manuell geschlossen werden, weil der Zeitpunkt der Garbage-Collection nicht vorhersehbar ist.

# Ein- und Ausgabe
## Datei-Streams

- Beispiel: ASCII-Datei zeichenweise mit **Reader**/**Writer** kopieren:

```
public class Copy {
  public static void main(String[] args) throws IOException {

    FileReader in  = new FileReader("Eingabe.txt");
    FileWriter out = new FileWriter("Ausgabe.txt");
    int c;

    while ((c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

- Anmerkungen: Da die meisten Dateisysteme 8-Bit basiert sind, wird jedes Zeichen in ein Unicode-Zeichen umgewandelt → ineffizient.
- Ungepuffertes Lesen und Schreiben → ineffizient.
- Für das Datei-Kopieren gibt es bereits die Klasse **Files** in **java.nio.file**.

- Beispiel: Binär-Datei byteweise mit **`InputStream`**/**`OutputStream`** kopieren:

```
public class Copy {
  public static void main(String[] args) throws IOException {

    FileInputStream  in  = new FileInputStream("Eingabe.dat");
    FileOutputStream out = new FileOutputStream("Ausgabe.dat");
    int c;

    while ((c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

- Ungepuffertes Lesen und Schreiben → ineffizient.
- **`Files`** aus **`java.nio.file`** kann auch Binärdateien kopieren

# Ein- und Ausgabe

## Hinweise zum Umgang mit Dateien

- Da Java eine plattformunabhängige Programmiersprache ist, sind im Umgang mit Dateinamen einige Punkte zu beachten.
- Die Klasse **`System`** des Paketes **`java.lang`** bietet die Methode **`getProperty("name")`**, um unabhängig von der Plattform Verzeichnisdienste in Anspruch nehmen zu können.

### Benutzereinstellungen

- Home-Verzeichnis des Benutzers:

  ```
  System.getProperty("user.home")
  ```

- Aktuelles Arbeitsverzeichnis des Benutzers:

  ```
  System.getProperty("user.dir")
  ```

### Dateitrennzeichen

- Pfadtrennzeichen sind auf verschiedenen Systemen unterschiedlich codiert: **`\`** unter Windows, **`/`** unter Unix → wird von den IO-Klassen automatisch angepasst.
- Die korrekte Darstellung wird zur Laufzeit mit

  ```
  System.getProperty("file.separator")
  ```

  ermittelt.

## Pfadtrennzeichen

- Sind in einer Environment-Variablen mehrere Pfade abgelegt, so ist das Trennzeichen auch systemabhängig.
- Mit `System.getProperty("path.separator")` kann die Einstellung auf der Plattform zur Laufzeit ermittelt werden.

## Zeilentrennzeichen

- Die Zeilen einer Textdatei werden durch verschiedene Trennzeichen beendet: `\n` unter Unix, `\r\n` unter DOS.
- Mit `System.getProperty("line.separator")` kann die Einstellung auf der Plattform zur Laufzeit ermittelt werden.

## Weitere Einstellungen

- Java unterstützt weitere Properties. Diese können mit `getProperties()` komplett gelesen werden. Nähere Informationen: siehe API-Dokumentation

# Ein- und Ausgabe

## Filter-Streams – Prinzip

- Viele Streams erwarten andere Streams als Konstruktor-Parameter:

```
FileReader     in  = new FileReader("Eingabe.txt");
BufferedReader bin = new BufferedReader(in);
```

- Die Methode öffnet einen **`BufferedReader`** auf **`in`**, welches selbst ein anderer Reader-Typ ist.
- Damit wird der Reader **`in`** in einen anderen Reader „eingepackt“.
- Das Programm liest von einem **`BufferedReader`**, welcher selbst von dem Reader **`in`** liest.
- Dieses geschieht, damit das Programm die Methode **`readLine`** von **`BufferedReader`** verwenden kann, um ganze Zeilen einer ASCII-Datei zu lesen.
- Das Einpacken von Readern und Writern oder Input- und OutputStreams ist eine sehr häufig eingesetzte Technik → Filter-Streams.

- Beispiel: ASCII-Datei zeilenweise mit **Reader**/**Writer** kopieren:

```
public class Copy {
  public static void main(String[] args) throws IOException {

    FileReader in  = new FileReader("Eingabe.txt");
    FileWriter out = new FileWriter("Ausgabe.txt");
    BufferedReader brIn  = new BufferedReader(in);
    PrintWriter    prOut = new PrintWriter(out);

    String line = brIn.readLine();

    while (line != null) {  // Abbruchbedingung: keine Zeile mehr
      prOut.println(line);
      line = brIn.readLine();
    }
    in.close();
    out.close();
  }
}
```

# Ein- und Ausgabe

## Filter-Streams – Beispiel

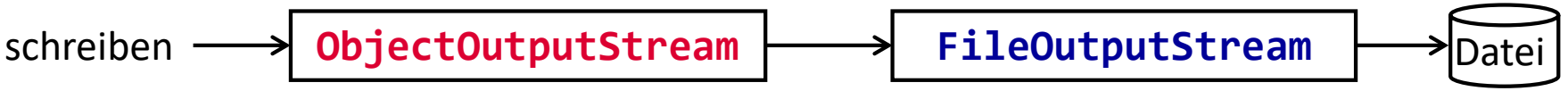
- **BufferedInputStream** zum gepufferten Lesen und Schreiben von Binärdaten:

```
public class BufferedCopy {
  public static void main(String[] args) throws IOException {

    InputStream  in  = new FileInputStream("Eingabe.dat");
    OutputStream out = new FileOutputStream("Ausgabe.dat");

    in  = new BufferedInputStream(in);
    out = new BufferedOutputStream(out);

    int c;

    while ((c = in.read()) != -1) {
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

- Zum Lesen und Schreiben kompletter Java-Objekte existieren die zwei Klassen **`ObjectInputStream`** und **`ObjectOutputStream`**.
- Objekte werden in einer bestimmten Art serialisiert, so dass sie später exakt wieder rekonstruiert werden können.
- Einsatzgebiete:
  - Kommunikation mit RMI
  - Leichtgewichts-Persistenz
  - ...

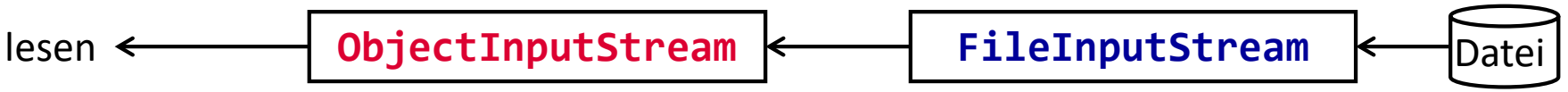
- Schreiben von Objekten auf einen **ObjectOutputStream**:

```
public class SerialWriter {

  public static void main(String[] args) {
    try {
      FileOutputStream out = new FileOutputStream("theTime");
      ObjectOutputStream str = new ObjectOutputStream(out);
      str.writeObject("Today");
      str.writeObject(new Date());
      str.flush();
      str.close();
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
  }
}
```



- Lesen der serialisierten Objekte:

```
public class SerialReader {
  public static void main(String[] args) {
    try {
        FileInputStream   in  = new FileInputStream("theTime");
        ObjectInputStream str = new ObjectInputStream(in);
        String today = (String) str.readObject();
        Date date = (Date) str.readObject();
        System.out.println(today + ": " + date);
        str.close();
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
    catch (ClassNotFoundException ex1) {
      System.err.println(ex1.getMessage());
    }
  }
}
```



- Anmerkungen:
  - Beim Lesen eines Objektes mit **`readObject`** können die folgenden Exceptions auftreten:
    - **`ClassNotFoundException`**: Die Klassendatei für das zu lesende Objekt ist nicht verfügbar.
    - **`InvalidClassException`**: Die Klasse hat keinen Default-Konstruktor, oder der Klassencode wurde nach der Serialisierung verändert, oder die Klasse hat unbekannte Datentypen.
    - **`StreamCorruptedException`**: Kontrollinformationen in den serialisierten Daten wurden verändert.
    - **`OptionalDataException`**: Der Stream enthält primitive Datentypen statt Objekte.
  - **`readObject`** liefert immer eine Referenz auf **`Object`** als Ergebnis zurück. Diese muss in den richtigen Typ umgewandelt werden.

## Serialisierung eigener Klassen

- Damit eigene Klassen serialisiert werden können, müssen sie das leere Interface `Serializable` implementieren → nur ein Flag!
- Eine nicht-serialisierbare Vaterklasse der serialisierbaren Klasse muss einen parameterlosen Konstruktor besitzen.
- Es ist nicht notwendig, Methoden selbst zu schreiben.
- Die Methode schreibt alle Informationen, die zur Wiederherstellung des Objektes notwendig sind:
  - Klasse des Objektes
  - Klassensignatur
  - Werte aller nicht-transienten und nicht-statischen Attribute.
  - Es werden auch alle referenzierten Objekte geschrieben → Baum oder Graph!

- Durch Angabe des Schlüsselwortes **`transient`** wird das entsprechende Attribut nicht mit serialisiert. Beispiel:

  ```
  transient int counter = 0;
  ```

  Eine weitere Bedeutung hat **`transient`** nicht.
- Der normale Serialisierungsmechanismus ist relativ langsam. Für spezielle Bedürfnisse kann er angepasst werden → nicht Bestandteil der Vorlesung.
- Warnung: Wird eine Klasse verändert, nachdem bereits Objekte von ihr serialisiert wurden, so können diese Daten nicht mehr einfach eingelesen werden. Es reicht schon, eine Methode hinzuzufügen!
- Zur Lösung des Problems kann man die Berechnung einer eindeutigen ID, die die Version einer Klasse (Prüfsumme) beinhaltet, selbst kontrollieren.

- Die bisherigen Beispiele haben die Fehlerbehandlung mehr oder weniger ignoriert.
- Wie sehen die Ein- und Ausgabe in einer Methode aus, die mögliche Ausnahmen selbst behandelt? Die bisherigen Behandlungen waren nicht ganz korrekt:

```
public class SerialWriter {

  public void write(MyClass object) {
    try {
      FileOutputStream out = new FileOutputStream("theTime");
      ObjectOutputStream str = new ObjectOutputStream(out);
      str.writeObject("Today");
      str.writeObject(new Date());
      str.flush();
      str.close();
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
  }
}
```

Wer schließt die Datei, wenn beim Schreiben ein Fehler auftritt? Niemand!

- Zweiter Versuch: Die Datei wird immer geschlossen (**finally**-Block!):

```
public class SerialWriter {

  public void write(MyClass object) {
    FileOutputStream out = null;
    try {
      out = new FileOutputStream("theTime");
      ObjectOutputStream str = new ObjectOutputStream(out);
      str.writeObject("Today");
      str.writeObject(new Date());
      str.flush();
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
    finally { // wird immer aufgerufen
      if (out != null) {
        out.close();  // Unvollständiger Code: löst auch IOException aus
      }
    }
  }
}
```

- Dritter Versuch:

```
public class SerialWriter {
  public void write(MyClass object) {
    FileOutputStream out = null;
    try {
      out = new FileOutputStream("theTime");
      ObjectOutputStream str = new ObjectOutputStream(out);
      str.writeObject("Today");
      str.writeObject(new Date());
      str.flush();
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
    finally { // wird immer aufgerufen
      if (out != null) {
        try {
          out.close();  // löst auch IOException aus
        }
        catch (IOException ex) {}
      }
    }
    // ...
```

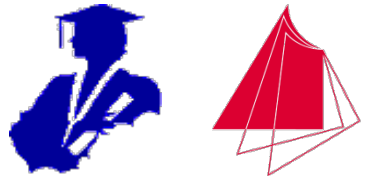
**Gruselcode!**

- Vierter Versuch: „try with resources“:

```
public class SerialWriter {
  public void write(MyClass object) {
    try (FileOutputStream out = new FileOutputStream("theTime")) {
      ObjectOutputStream str = new ObjectOutputStream(out);
      str.writeObject("Today");
      str.writeObject(new Date());
      str.flush();
    }
    catch (IOException ex) {
      System.err.println(ex.getMessage());
    }
  }
}
```

- Die im **`try`**-Block geöffnete Ressource wird automatisch wieder geschlossen.
- Werden mehrere Ressourcen geöffnet, dann werden diese durch Semikolons getrennt aufgeführt.
- Die Ressource muss die Schnittstelle **`java.lang.AutoCloseable`** implementieren → geht also auch für eigene Klassen.

- Einige weitere Filter:
  - **`GZIPInputStream`**/**`GZIPOutputStream`**: Lesen und Schreiben von GZIP-Archiven
  - **`ZIPInputStream`**/**`ZIPOutputStream`**: Lesen und Schreiben von ZIP-Archiven
  - **`JarInputStream`**/**`JarOutputStream`**: Lesen und Schreiben von JAR-Archiven
  - **`PushbackInputStream`**: Stream, bei dem gelesene Bytes wieder „ungelesen“ werden können.
  - **`XMLDecoder`**/**`XMLEncoder`**: Schreiben und Lesen der Objekte im XML-Format
- Weitere Datenquelle:
  - **`URL`**: Methode **`openStream`** liefert einen **`InputStream`**, um binäre Daten von der URL zu lesen.
- Und wie sieht es mit speziellen Dateiformaten aus (Bilder, Videos, Excel-Dateien)?
- Dafür gibt es entweder spezielle Bibliotheken oder man schreibt das selbst...



- Auch die aus den Datenstrukturen bekannte Stream-API kann bei Dateien sehr gut verwendet werden.
- Dazu werden Klassen aus dem Paket **`java.nio.files`** benötigt:
  - **`FileSystems`**: kapselt Informationen über das verwendete Dateisystem
  - **`Files`**: erlaubt den Zugriff auf Dateien, Verzeichnisse, …
- Beispiel, um die Anzahl der Dateien mit der Endung **`exe`** in einem Verzeichnis zu ermitteln:

```
Stream<Path> str = Files.list(FileSystems.getDefault().getPath("e:\\Downloads"));
long count = str.filter(x -> x.toString().endsWith(".exe")).count();
```

- Beispiel, um die Gesamtgröße aller Dateien mit der Endung exe in einem Verzeichnis zu ermitteln:

```
Stream<Path> str = Files.list(FileSystems.getDefault().getPath("e:\\Downloads"));
long sum = str.filter(x -> x.toString().endsWith(".exe"))
              .mapToLong(x -> x.toFile().getTotalSpace()).sum();
```

- Natürlich gibt es hier auch noch viel mehr Möglichkeiten (z.B. das zeilenweise Auslesen von Texten aus einer ASCII-Datei).