

# FOP Recap #12



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

IO



# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wiederholung: Wann ist etwas generisch?

- Generische Klassen

- Generische Methoden

- Nicht generische Methoden

Grundlagen

File-IO



## Definition – Generizität:

Eine Methode oder Klasse ist genau dann generisch, wenn sie mindestens einen Typparameter besitzt.

In einer Klasse/Methode können generische Klassen/Methoden vorkommen, ohne dass diese generisch sein muss, insbesondere ist ien Methode, in der nur Wildcards vorkommen, nicht generisch.

# Wiederholung: Wann ist etwas generisch?

## Generische Klassen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class MyClass {  
2     // ...  
3 }
```

```
1 public class MyGenericClass <T> {  
2     // ...  
3 }
```

```
1 public class AnotherGenericClass <I, O> {  
2     // ...  
3 }
```

# Wiederholung: Wann ist etwas generisch?

## Generische Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public void nonGenMethod() {  
2     // ...  
3 }
```

```
1 public <T extends Number> List<T> genericMethod(T number) {  
2     // ...  
3 }
```

```
1 public <I, O> List<O> genericMethod1(List<? extends I> list) {  
2     // ...  
3 }
```

# Wiederholung: Wann ist etwas generisch?

## Nicht generische Methoden



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public List<String> nonGenMethod() {  
2     // ...  
3 }
```

```
1 // T sei in Klasse definiert  
2 public void nonGenMethod1(List<T> number) {  
3     // ...  
4 }
```

```
1 public List<Number> nonGenMethod2(List<? super Number> list) {  
2     // ...  
3 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wiederholung: Wann ist etwas generisch?

Grundlagen

Was ist IO?

Java

Habe ich da Stream gehört?

Klasse System

File-IO



- IO steht für Input/Output
- Zum Beispiel:
  - ▣ Konsolenausgaben
  - ▣ Lesen und Schreiben von Dateien
  - ▣ Unterschiedliche Typen:
    - Textdateien (z.B. .txt)
    - Bilddateien (z.B. .png, .jpg)
    - Sounddateien (z.B. .ogg, .mp3)
    - Videodateien (z.B. .mp4)
    - ....





- Die JVM stellt sehr viel Funktionalität bereit
- Man muss komplizierte Dinge nicht selber schreiben
- Typische Klassen, mit denen man hier arbeitet:
  - File
  - InputStream
  - OutputStream
  - Reader
  - Writer
  - IOException
  - ...

# Grundlagen

Habe ich da Stream gehört?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- InputStream und OutputStream gab es zuerst
- Teilen sich dieselbe Grundidee: Funktionieren wie ein Fließband
- Haben jedoch in erster Linie nichts miteinander zu tun



- Kann nicht instantiiert werden, hat nur Klassenattribute/-Methoden
- Die JVM sorgt für Unabhängigkeit vom Betriebssystem
- Stellt Methoden bereit, um mit der Umgebung zu interagieren, z.B. `getenv` für Umgebungsvariablen
- Hat die statischen Attribute `in`, `out` und `err`

# Grundlagen

## Klasse `System` – `InputStream` und `OutputStream`



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Repräsentieren Ein- bzw. Ausgabestreams an Bytes
- Viele Subklassen in der Standardbibliothek
- Hier vor allem über die Klasse `System` relevant
- Auch in anderen Kontexten nutzbar, siehe Oracle-Docs

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Wiederholung: Wann ist etwas generisch?

Grundlagen

File-IO

- Grundlagen

- FileReader

- BufferedReader

- FileWriter

- BufferedWriter

- Exceptions



Grobe Vereinfachung:

- In Dateien speichert man Daten
- Daten bestehen aus Bytes
- Dateiformate legen Codierung der Daten fest
- Unterscheidung zwischen Binärdateien und Textdateien
  - ▣ Textdateien: Enthalten Byte-Sequenzen, die als Textzeichen interpretiert werden können
  - ▣ Binärdateien sind alles andere
  - ▣ Bei beiden Arten ist Codierung relevant!

Wir behandeln hier nur das Größte, die Standardbibliothek stellt sehr viel Funktionalität bereit  
-> Oracle Docs

- Klasse für das Lesen von Textdateien
- Hat verschiedene read-Methoden für das Einlesen von Zeichen

```
1 // unhandled exceptions!
2 File file = new File("/home/wolf/Documents/myDocument.md");
3 FileReader fileReader = new FileReader(file);
4 int c;
5 while((c = fileReader.read()) != -1)
6     System.out.print(Character.toString(c));
7 // c is now -1
8 fileReader.close();
9 // don't use fileReader from now on!
10 // the closing isn't optimal, see the chapter on exceptions
```



- Kann man genau so wie `FileReader` verwenden
- Abstrahiert einen Reader durch Verwendung eines Puffers (Buffer)
- Geht somit z.B. besser mit Systemressourcen um
- Stellt weitere Methoden bereit
- Sollte man fast immer verwenden, wenn man Zeichen irgendwoher einliest

```
1 // unhandled exceptions!
2 BufferedReader bufferedReader =
3     new BufferedReader(new FileReader("myDocument.md"));
4 bufferedReader.lines().forEach(System.out::println);
5 bufferedReader.close();
6 // don't use bufferedReader from now on!
7 // the closing isn't optimal, see the chapter on exceptions
```



- Klasse zum Schreiben von Textdateien
- Hat verschiedene write-Methoden für das Schreiben von einzelnen oder mehreren Zeichen
- Warnung: Manches Verhalten ist plattformabhängig!

```
1  // unhandled exceptions!
2  File file = new File("myFile.md");
3  FileWriter fileWriter = new FileWriter(file, false);
4  fileWriter.write("line 1");
5  fileWriter.append(System.lineSeparator())
6      .append("line ")
7      .append('2')
8      .close();
9  // don't use fileWriter from now on!
10 // the closing isn't optimal, see the chapter on exceptions
```



- Selbes Puffer-Konzept wie BufferedReader
- Kann man dank Interfaces und Vererbung auch genau wie den Writer verwenden
- flush-Operation: Schreibe den Puffer mithilfe des Writers, passiert auch automatisch
- Sollte man fast immer verwenden, wenn man Zeichen irgendwo schreibt

```
1 // unhandled exceptions!
2 BufferedWriter bufferedWriter =
3     new BufferedWriter(new FileWriter("myFile.md", false));
4 for (String str : IntStream.range(1, 1001)
5     .mapToObj(i -> "line " + i).toList()) {
6     bufferedWriter.append(str);
7     bufferedWriter.newLine();
8 }
9 bufferedWriter.close(); // don't use bufferedWriter from now on!
10 // the closing isn't optimal, see the chapter on exceptions
```



- Superklasse von allen Exceptions, die während IO-Operationen geworfen werden können
- Hat Subtypen wie z.B. `FileNotFoundException`
- Jede Lese/Schreib-Operation kann eine `IOException` werfen
- Nicht von `RuntimeException` abgeleitet -> Muss behandelt werden!



- Wenn Exceptions geworfen werden, werden Ressourcen potentiell nicht geschlossen!
- Kann zu Problemen führen, sollte man vermeiden

```
1  try {  
2      FileReader fileReader = new FileReader("myDocument.md");  
3      BufferedReader bufferedReader =  
4          new BufferedReader(fileReader);  
5      String firstLine = bufferedReader.readLine();  
6      // ...  
7      // not executed if an exception is thrown  
8      bufferedReader.close();  
9  }  
10 catch(IOException e) {  
11     e.printStackTrace();  
12 }
```



- finally-Block wird immer nach try-Block oder catch-Block ausgeführt

```
1  BufferedReader reader = null;
2  try {
3      reader = new BufferedReader(new FileReader("myDocument.md"));
4      String firstLine = reader.readLine();
5      // ...
6  }
7  catch(IOException e) {
8      e.printStackTrace();
9  }
10 finally {
11     // TODO: close reader
12 }
```



- Probleme treten insbesondere dann auf, wenn mehrere Ressourcen zu verwalten sind

```
1 // ...
2 finally {
3     // close reader... uh oh
4     if(reader != null) {
5         try {
6             reader.close();
7         }
8         catch(IOException e) {
9             e.printStackTrace();
10        }
11    }
12 }
```



- Ressourcen am Anfang von Block deklarieren und initialisieren
- Compiler übernimmt den Rest

```
1 try(FileReader fileReader = new FileReader("myDocument.md");
2     BufferedReader bufferedReader =
3         new BufferedReader(fileReader)) {
4     String firstLine = bufferedReader.readLine();
5     // ...
6 }
7 catch(IOException e) {
8     e.printStackTrace();
9 }
```



---

# Live-Coding!