Fortgeschrittene Programmiertechniken WS24/25

Universität Duisburg-Essen
Fakultät für Informatik
Abteilung für Software Engineering
Intelligente Systeme
Josef Pauli

Vorlesung

- Dozent der Vorlesung: Fatih Özgan → fatih.oezgan@uni-due.de
- Wöchentlich in Präsenz, dienstags 10:15 - 11:45 Uhr in LB 131
- Vorlesungsfolien, Programm-Code werden publiziert.
- Diskussionsforum im Moodle-Kurs

Übung

- Übungsleiter: Fatih Özgan, Jonathan Dreisvogt → jonathan.nepomuk.dreisvogt@stud.uni-due.de
- Erste Übung 18.10.2024
- Zeitfenster:
  - Freitag 08:00 Uhr - 10:00 Uhr
  - Freitag 10:00 Uhr - 12:00 Uhr
  - Freitag 12:00 Uhr - 14:00 Uhr
- Zur besseren Verteilung Anmeldung zur ersten Übung über Moodle
- Es gibt keine Bonuspunkte

Wichtig!

Programmieren lernt man nur durch Praxis!
Hoher Programmieranteil in der Klausur!

Übung: Infrastruktur

- In LF 257 gibt es 11 PCs, und je Arbeitsplatz ca. 2 Sitzplätze, also ca. 20-22 Sitzplätze
- Arbeit in Kleingruppen möglich
- Installiert wurde in LF 257:
  - IntelliJ IDEA Community Edition, idealC-2019.3.5
    https://www.jetbrains.com/de-de/idea/download/other.html
  - Bellsoft JDK 11
    https://bell-sw.com/pages/downloads/#/java-11-lts
- Empfohlen wird diese Installation auch auf eigenen Rechnern.
- Anleitung zur Installation im Moodle-Kurs im Bereich Arbeitsumgebung.
- WICHTIG: Falls Sie Ihren eigenen Rechner verwenden wollen, sollten Sie die Installation bereits vor der ersten Übung durchgeführt haben.

Vorlesungsinhalt

- Java Einführung
- Objektorientierung
- Kopieren von Objekten und Serialisierung
- Entwurfsmuster
- Nebenläufige Programmierung (Threads)
- Datenbankanbindungen
- Netzwerkprogrammierung (TCP, UDP)
- Grundlagen GUI-basierte Systeme (JavaFX und MVC)

Erste Klausurhinweise

- Die Klausur wird die Vorlesung und die Übung zum Inhalt haben.
- Wir empfehlen dringend die Übungen aktiv zu bearbeiten.
- Nur dadurch befähigen Sie sich, die Aufgaben in der Klausur gut und schnell zu bearbeiten.

1. Einführung Java & Objektorientierung

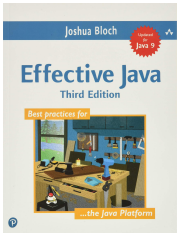
Universität Duisburg-Essen
Fakultät für Informatik
Abteilung für Software Engineering
Intelligente Systeme
Josef Pauli

1.1 Motivation

Disclaimer

- Im Rahmen dieser Vorlesung wird die Programmierung in Java in der **Breite** vorgestellt
- Sprachfeatures werden nicht in Gänze/Tiefe behandelt
- Demonstration von gängigen Anwendungsszenarien
- Das Studium von **weiterführender** Literatur und **praktische** Übungen werden empfohlen

Primo Link



Primo Link

Primo Link

Primo Link

Java Version und IDE

Java Version

- In der Vorlesung wird Java 13 von Oracle unter Windows 10 verwendet
- Wahl steht frei, aber mindestens Java 8
- Oracle JDK oder Bellsoft JDK

IDE

- In der Vorlesung wird IntelliJ IDEA verwendet
- Eclipse, NetBeans ...

Ziel

Hello World im Detail!

Python

```
print("Hello World");
```

Ziel

Hello World im Detail!

Python

```
print("Hello World");
```

Java

```
public class SomeClass {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Ziel

Hello World im Detail!

Python

```
print("Hello World");
```

Java

```
public class SomeClass {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Warum neue Programmiersprache!?

Warum neue Programmiersprache?

Akademische Gründe:

- Neues Paradigma $\rightarrow$ Objektorientierte Programmierung
- Statische Typisierung

Warum neue Programmiersprache?

Akademische Gründe:

- Neues Paradigma → Objektorientierte Programmierung
- Statische Typisierung

Praktische Gründe:

- Große Projekte verwenden **eher selten** Python
- Java wird in der Industrie **intensiv** eingesetzt
- **Standard** für Business Anwendungen
- Mehr Kontrolle und Einsicht in die internen Abläufe

Warum neue Programmiersprache?

- Es gibt nicht **die** beste Programmiersprache!
- Es kommt **immer** auf die Problemstellung an!

Warum neue Programmiersprache?

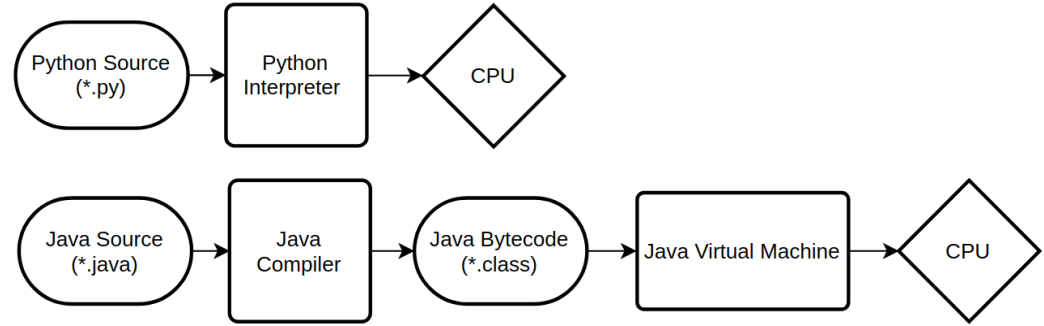
- Es gibt nicht **die** beste Programmiersprache!
- Es kommt **immer** auf die Problemstellung an!

When in Rome, do as the Romans do!

1.2 Unterschiede Java/Python

(Ausführung und Strukturierung)

Unterschied: Ausführung



- Python Quellcode wird von einem **Interpreter** ausgeführt
- Java Quellcode wird von einem **Java-Compiler** nach **Bytecode** übersetzt und dann von einer **Java Virtual Machine** ausgeführt → Optimierungsmöglichkeit
- **Just-in-time-Kompilierung** wird intensiv verwendet

Strukturierung

Bei der **Strukturierung durch Einrückung** haben Leerzeichen und neue Zeilen eine semantische Bedeutung.

```
i = 1
s = 0
while i <= 100:
  s += i
  ++i
print("Die Summe lautet: ", s)
```

Strukturierung

Bei der **Blockstrukturierung** werden Blöcke und Anweisungen durch **Schlüsselwörter** oder **Sonderzeichen** gekennzeichnet.

```
i = 1;
s = 0;
while (i <= 100) {
  s += i;
  ++i;
}
```

Strukturierung

Bei der **Blockstrukturierung** werden Blöcke und Anweisungen durch **Schlüsselwörter** oder **Sonderzeichen** gekennzeichnet.

```
i = 1;
s = 0;
while (i <= 100) {
  s += i;
  ++i;
}
```

Eine sinnvolle Einrückung wird dennoch empfohlen!

```
i=1;s=0;while(i<=100){s+=i;++i;}
```

Typisierung

Typisierung

Dynamische Typisierung

Der Datentyp von Variablen kann sich ändern.

```
variable = "I am a string!" # OK
variable = 1234 # OK
```

Typisierung

Dynamische Typisierung

Der Datentyp von Variablen kann sich ändern.

```
variable = "I am a string!" # OK
variable = 1234 # OK
```

Statische Typisierung

Der Datentyp von Variablen wird schon während Deklaration festgelegt. Er kann sich im weiteren Verlauf **nicht** ändern.

```
String variable = "I am a string!"; // OK
variable = 123; // # Error
int variable = 123; // # Error
```

Typisierung

Statische Typisierung

Die **statische Typisierung** gilt auch für **Funktionsparameter** und **Funktionsrückgabewerte**.

```
String function(int param1, float param2) {
  return "I Am a String";
}
```

Typisierung

Statische Typisierung

Die **statische Typisierung** gilt auch für **Funktionsparameter** und **Funktionsrückgabewerte**.

```
String function(int param1, float param2) {
  return "I Am a String";
}
```

Wenn es keinen Rückgabewert gibt, dann wird void verwendet.

```
void procedure(int param) {
  ...
}
```

1.3 Einstieg Objektorientierung

Prozedural

Bisher wurde **imperativ** und **prozedural** programmiert.

- Vorgabe von Anweisungen und ihrer Reihenfolge
  - → Verwendung von Variablen, Operatoren, Fallunterscheidungen, Schleifen, einfache statische Methoden

- Zerlegung von Programmen in überschaubare Teile
  - → Verwendung von **Prozeduren**
  - → Das **'Wie'** stand im Vordergrund
  - → Welche **Schritte/Funktionalitäten** sind notwendig?

Prozedurale Programmierung

Beispiel:

Für ein Betriebsystem soll eine Komponente entwickelt werden, die die Dateiverwaltung übernimmt. Welche **Funktionalitäten** sind notwendig?

Prozedurale Programmierung

Beispiel:

Für ein Betriebsystem soll eine Komponente entwickelt werden, die die Dateiverwaltung übernimmt. Welche **Funktionalitäten** sind notwendig?

```
boolean createFile(String name);
boolean createFolder(String name);
void setFlags(String filename, int flags);
boolean exists(String name);
```

Objektorientierung

Bei der **Objektorientierung** stehen **Objekte** im Vordergrund. Neben dem **'Wie'** wird nun auch das **'Was'** berücksichtigt!

Beispiel:

Für ein Betriebsystem soll eine Komponente entwickelt werden, die die Dateiverwaltung übernimmt. Welche **Objekte** gibt es?

Objektorientierung

Bei der **Objektorientierung** stehen **Objekte** im Vordergrund. Neben dem **'Wie'** wird nun auch das **'Was'** berücksichtigt!

Beispiel:

Für ein Betriebsystem soll eine Komponente entwickelt werden, die die Dateiverwaltung übernimmt. Welche **Objekte** gibt es?

Objekte:

- Dateien
- Ordner
- Zugriffsrechte

Objektorientierung

Bei der **Objektorientierung** stehen **Objekte** im Vordergrund. Neben dem **'Wie'** wird nun auch das **'Was'** berücksichtigt!

- Formell ausgedrückt: Die Architektur der Software wird an den **Grundstrukturen** der betreffenden Anwendungsdomäne ausgerichtet.
- Um diese **Denkweise** anwenden zu können, sind Konzepte wie **Klassen, Vererbung, Datenkapselung, Polymorphie** und **Komposition** notwendig.

Unterschied Prozedural und Objektorientierung

In Objektorientierung sind Attribute und Verhaltensweisen eingebettet in einzelne Objekte, während in prozeduraler Programmierung Attribute und Verhaltensweisen getrennt werden.

Person

- Attribute (Eigenschaften)
  - Augenfarbe
  - Größe
  - Geschlecht
- Verhalten (Methoden)
  - Laufen
  - Atmen
  - Sprechen

Klassen

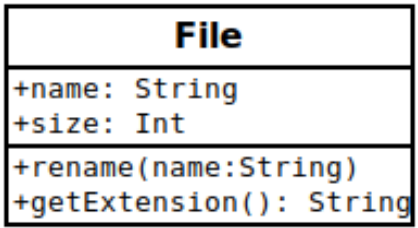
Was ist eine **Klasse**?

- Unter einer **Klasse** versteht man ein **abstraktes Modell (Schablone)** für eine Reihe von ähnlichen Objekten.
- Eine **Klasse** kann **Eigenschaften (Attribute)** und **Verhaltensweisen (Methoden)** aufweisen.

Klassen

Was ist eine **Klasse**?

- Unter einer **Klasse** versteht man ein **abstraktes Modell (Schablone)** für eine Reihe von ähnlichen Objekten.
- Eine **Klasse** kann **Eigenschaften (Attribute)** und **Verhaltensweisen (Methoden)** aufweisen.




```
class File {
  // Attribute
  int size;
  String name;

  // Methoden
  void rename(String name){...};
  String getExtension(){...};
}
```

Objekte

Was sind **Objekte**?

- Objekte sind konkrete Ausprägungen (Instanzen) einer Klasse
- Sie werden durch new erzeugt (Instanziierung)
- Auf die Attribute und Methoden kann mit dem **Punkt-Operator** zugegriffen werden

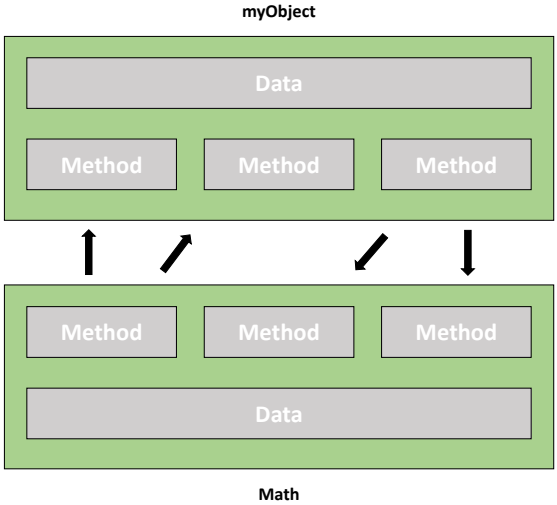
```
File textFile = new File();
textFile.name = "readme.txt";
textFile.size = 27;
textFile.getExtension();

File videoFile = new File();
videoFile.name = "LOTR.mp4";
videoFile.size = 10037265;
videoFile.rename("Lord of the Rings.mp4");
```

Objekte

Mit dem var Schlüsselwort kann man sich Schreibarbeit sparen. Die **statische Typisierung** findet weiterhin Anwendung!

```
var textFile = new File();
var videoFile = new File();
var myString = "I am a String!";
```

Kommunikation zwischen Objekten



Zugriffsmodifikatoren

Durch **Zugriffsmodifikatoren** wird festgelegt, wer auf die **Methoden/Attribute** der Objekte zugreifen darf.

```
class File {
  private int size;
  private String name;

  public void rename(String name) {
    this.name = name;
  }
}
```

```
File file = new File();
file.rename("readme.txt"); // OK
file.name = "lotr.mp4"; // ERROR
```

Zugriffsmodifikatoren

- Klassen sind `public` oder `package-private`, d.h. ohne Zugriffsmodifikator sein.
- Member einer Klasse sind `public`, `package-private`, `protected` oder `private`.

Modifier	Class	Package	Subclass	World
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
`private`	Yes	No	No	No

Datenkapselung

Durch die **Zugriffsmodifikatoren** können **Getter** und **Setter** zur **Datenkapselung** und Einhaltung von **Invarianten** umgesetzt werden.

```
class File {
  private int size;

  public void setSize(int size) {
    if (size < 0) {
      size = 0;
    }
    this.size = size;
  }
  public int getSize() {
    return size;
  }
}
```

Konstruktoren

Der **Konstruktor** wird aufgerufen, wenn ein neues Objekt mittels new erstellt wird. Alle **Konstruktoren** haben den gleichen Namen, wie die dazugehörige Klasse.

```
class File {
  private int size;
  private String name;

  public File(String name) {
    this.name = name;
  }
}
```

```
File file = new File("readme.txt");
```

Konstruktoren

Standardkonstruktor

Falls kein Konstruktor definiert wird, legt der Compiler automatisch einen Standardkonstruktor (*engl. default constructor*) an.

Konstruktoren

Standardkonstruktor

Falls kein Konstruktor definiert wird, legt der Compiler automatisch einen Standardkonstruktor (*engl. default constructor*) an.

$\rightarrow$ Best Practice: Explizit einen Standardkonstruktor anlegen.

Konstruktoren

Standardkonstruktor

Falls kein Konstruktor definiert wird, legt der Compiler automatisch einen Standardkonstruktor (*engl. default constructor*) an.

$\rightarrow$ Best Practice: Explizit einen Standardkonstruktor anlegen.

Überladener/Parametrisierter Konstruktor

Ein Konstruktor kann genau wie eine Methode Parameter besitzen, die zur Erzeugung der Objekte verwendet werden. Es können mehrere Konstruktoren definiert werden, die sich in ihrer Signatur unterscheiden müssen.

Konstruktoren

Standardkonstruktor

Falls kein Konstruktor definiert wird, legt der Compiler automatisch einen Standardkonstruktor (*engl. default constructor*) an.

→ Best Practice: Explizit einen Standardkonstruktor anlegen.

Überladener/Parametrisierter Konstruktor

Ein Konstruktor kann genau wie eine Methode Parameter besitzen, die zur Erzeugung der Objekte verwendet werden. Es können mehrere Konstruktoren definiert werden, die sich in ihrer Signatur unterscheiden müssen.

→ Parametrisierter Konstruktor = Parameterloser Konstruktor + Setter-Methoden

Konstruktoren

```
class Person {
  // Attribute
  private String name;
  private int age;

  // Konstruktoren
  public Person() {} // Standardkonstruktor

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

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

```
  Person person1 = new Person();
  Person person2 = new Person("Max");
  Person person3 = new Person("Max", 18);
```

Generische Klassen

Oftmals gibt es mehrere Klassen, die sich nur durch den verwendeten Datentypen unterscheiden

```
public class Point2i {
  int x;
  int y;
}
```

```
public class Point2f {
  float x;
  float y;
}
```

Generische Klassen

Oftmals gibt es mehrere Klassen, die sich nur durch den verwendeten Datentypen unterscheiden

```
public class Point2i {
  int x;
  int y;
}
```

```
public class Point2f {
  float x;
  float y;
}
```

Viel Schreibarbeit und fehleranfällig ...

Generische Klassen

Generische Klassen haben **Typstellvertreter**. Der konkrete Typ wird erst bei der **Instanziierung** festgelegt.

```
public class Point2<T> {
  T x;
  T y;
}
```

```
Point2<Integer> p1 = new Point2<Integer>();
Point2<Float> p2 = new Point2<Float>();
```

Es dürfen **keine** primitiven Datentypen verwendet werden.

Statics

- Statische Methoden oder Attribute gehören zur Klasse und nicht zum Objekt
  $\rightarrow$ Zugriff von außerhalb über Klassenname
- Alle Objektinstanzen haben Zugriff und teilen sich die Daten

```
class File {
  private static int createdFiles = 0;
  public File() {
    ++createdFiles;
  }
  public static getCreatedFiles()  { return createdFiles; };
}
```

```
File textFile = new File();
File videoFile = new File();
System.out.println(File.getCreatedFiles()); // 2
```

Standardausgabe

Die Standardausgabe erfolgt über **statisches** Attribut.

```
System.out.println("Hello World");
```

Standardausgabe

Die Standardausgabe erfolgt über **statisches** Attribut.

```
System.out.println("Hello World");
```

- `System`: Klasse des `java.lang` Pakets
- `out`: Statische Instanzvariable der Klasse `PrintStream`
- `println`: Objektmethode der Klasse `PrintStream`

1.4 Packages

Packages

- Jede ***.java** Datei darf nur eine Klasse beinhalten
- Klassen können in **packages** organisiert werden
- **Paketstruktur** wird durch eine **Ordnerstruktur** abgebildet

```
package MyPackage;
public class MyClass {}
```

```
package MyPackage;
public class MyOtherClass {}
```

Packages

- Jede ***.java** Datei darf nur eine Klasse beinhalten
- Klassen können in **packages** organisiert werden
- **Paketstruktur** wird durch eine **Ordnerstruktur** abgebildet

```
package MyPackage;
public class MyClass {}
```

```
package MyPackage;
public class MyOtherClass {}
```

Wird eine Klasse aus einem anderen Paket verwendet, so muss der komplette Pfad angegeben werden.

```
MyPackage.MyClass object = new MyPackage.MyClass();
```

Packages

Mit import können Klassen importiert werden, sodass die komplette Pfadangabe **entfallen** kann.

```
import MyPackage.MyClass;  // Importiert eine Klasse
import MyPackage.*; // Importiert alle Klassen aus MyPackage
```

Packages

- Klassen innerhalb von Paketen haben ebenfalls **Zugriffsmodifikatoren**
- **Standardmäßig** können Klassen nur innerhalb des selben Paketes verwendet werden
- public Klassen können auch von außerhalb des eigenen Paketes verwendet werden

Packages

- Klassen innerhalb von Paketen haben ebenfalls **Zugriffsmodifikatoren**
- **Standardmäßig** können Klassen nur innerhalb des selben Paketes verwendet werden
- public Klassen können auch von außerhalb des eigenen Paketes verwendet werden

```
package MyPackage;
public class MyClass {}
```

```
package MyPackage;
class MyOtherClass {}
```

```
var object1 = new MyPackage.MyClass(); // Ok
var object2 = new MyPackage.MyOtherClass(); // Error
```

1.5 Variablentypen

Variable

Variable

Ein **Behälter (Speicherbereich)** der mit einem **Namen** versehen wurde und zur **Laufzeit** verändert werden kann.

Variable

Variable

Ein **Behälter (Speicherbereich)** der mit einem **Namen** versehen wurde und zur **Laufzeit** verändert werden kann.

Variablentyp: **Wertevariable**

- ▷ Der Wert steht **direkt** in der Variable.
- ▷ Bei Zuweisung wird der **Wert** kopiert (dupliziert, gespeichert)

Variable

Variable

Ein **Behälter (Speicherbereich)** der mit einem **Namen** versehen wurde und zur **Laufzeit** verändert werden kann.

Variablentyp: **Wertevariable**

- Der Wert steht **direkt** in der Variable.
- Bei Zuweisung wird der **Wert** kopiert (dupliziert, gespeichert)

Variablentyp: **Referenz-Variable**

- In der Variable steht ein **Verweis (Speicheradresse)**, der auf den **eigentlichen** Wert zeigt.
- Bei Zuweisung wird **nur** die Referenz kopiert
- Mehrere Referenz-Variablen können über die Referenz auf die **selben** Daten zeigen

Primitive Datentypen

Primitive Datentypen

- Werden in **Wertevariablen** gespeichert
- Bilden Zahlen, Zeichen oder logische Werte ab
  - → Beinhalten einzelne Werte
- Haben einen **kleinen** Anfangsbuchstaben
- Es können **keine** weiteren primitiven Datentypen vom Benutzer hinzugefügt werden

```
int i = 3;
float f = 3.141;
boolean b = true;
byte u = 102;
```

Primitive Datentypen

Mit = bzw. == wird der Inhalt der Variablen, also der **Wert**, zugewiesen bzw. verglichen.

```
int i = 3;
int j = 3;
System.out.println(i == j); // true
int k = j;
System.out.println(i == k); // true
```

Referenz Datentypen

Objekt-Datentypen

- Werden in **Referenzvariablen** gespeichert
  → Können auf null zeigen
- Haben jeweils als Typ eine **Klasse**
- Klassen haben einen **großen** Anfangsbuchstaben
- Mit new wird im Hauptspeicher das Objekt einer Klasse angelegt und dessen Adresse in der Referenzvariable gespeichert
- Objekt-Datentypen sind aus anderen Datentypen zusammengesetzt

```
File fileA = new File();
File fileC = null;
```

Referenz Datentypen

Achtung!

Mit = bzw. == wird der Inhalt der Variablen, also die **Referenz**, und **nicht** die Objekte zugewiesen bzw. verglichen.

```
File file1 = new File("readme.txt");
File file2 = new File("readme.txt");
System.out.println(file1 == file2); // false
file2 = file1;
System.out.println(file1 == file2); // true
file2.rename("LOTR.mp4");
System.out.println(file1 == file2); // true
```

Referenz Datentypen

Objekte werden mit **equals** auf Gleichheit getestet.

```
File file1 = new File("readme.txt");
File file2 = new File("readme.txt");
System.out.println(file1 == file2); // false
System.out.println(file1.equals(file2)); // true
```

Referenz Datentypen

Objekte werden mit **equals** auf Gleichheit getestet.

```
File file1 = new File("readme.txt");
File file2 = new File("readme.txt");
System.out.println(file1 == file2); // false
System.out.println(file1.equals(file2)); // true
```

- Für eigene Klassen muss **equals** selber implementiert werden
- Es sollte sich an den equals contract gehalten werden
- Die Standardimplementierung von **equals** verwendet nur ==
- In der Dokumentation nachschauen!

Referenz Datentypen

Das Kopieren von Objekten wird umfangreich im Kapitel **Serialisierung** behandelt. Viele Klassen bieten allerdings die Methode **clone** an.

```
File file1 = new File("readme.txt");
File file2 = file1.clone();
System.out.println(file1 == file2); // false
System.out.println(file1.equals(file2)); // true
```

Parameterübergabe

Parameterübergabe

Bei der Parameterübergabe an Funktionen sollte auf den Datentyp geachtet werden:

- Bei **Primitive Datentypen** werden die Werte kopiert
  → **Call-by-Value**
- Bei **Referenz-Datentypen** wird nur die Referenz kopiert
  → **Call-by-Reference**

```
void method(int integer, // Call by Value
            File file) // Call by Reference
```

1.6 Einschub Speicherlayout

Vereinfachtes Speicherlayout

Stack

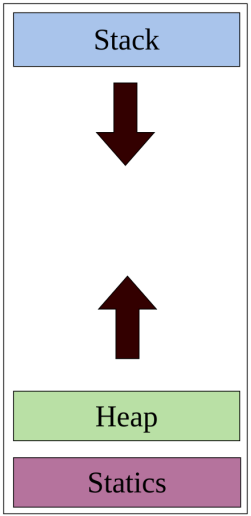
- Rücksprungadressen und lokale Variablen
- Wächst nur in eine Richtung und ist sehr schnell

Heap

- Objekte und die dazugehörigen Attribute
- Schwieriger in der Verwaltung

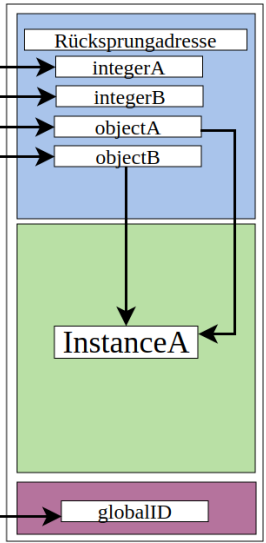
Statics

- **statische** Variablen und Konstanten
- Steht zur **Compilezeit** fest



Vereinfachtes Speicherlayout

```
public class MainClass {
    public static void main(String[] args) {

        int integerA = 5;
        int integerB = integerA;


        Object objectA = new Object();
        Object objectB = objectA;
    }

    static int globalID = 12345;
}
```

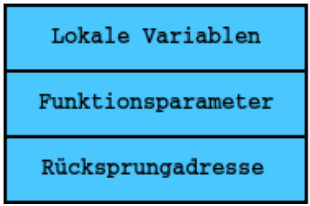


```
int main() {
  f1(123);
}
```

```
void f1(int param1) {
  int local1 = 456;
}
```

```c
int main() {
  f1(123);
}
```

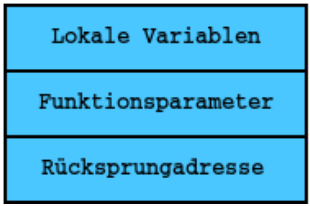
```c
void f1(int param1) {
  int local1 = 456;
}
```



```c
int main() {
  f1(123);
}
```

```c
void f1(int param1) {
  int local1 = 456;
  f2();
}
```

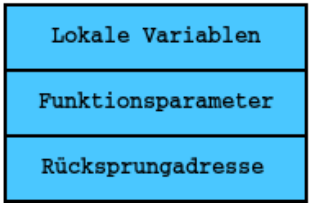
```c
void f2() {
  int local2 = 3;
}
```



```c
int main() {
  f1(123);
}
```

```c
void f1(int param1) {
  int local1 = 456;
  f2();
}
```

```c
void f2() {
  int local2 = 3;
}
```

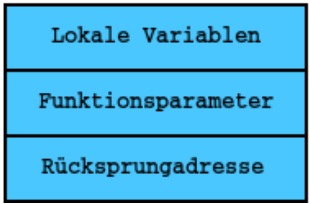


Die Lebensdauer von **lokalen** Variablen auf dem **Stack** ist auf den **Scope** begrenzt!

```c
int main() {
  f1(123);
}
```

```c
void f1(int param1) {
  int local1 = 456;
  f2();
}
```

```c
void f2() {
  int local2 = 3;
}
```



Daten mit **dynamischer** Lebensdauer können nicht auf dem Stack liegen!

1.7 final und Immutability

final

Mit final deklarierte Variablen können nur **einmal** einen Wert zugewiesen bekommen.

```
final float pi = 3.141;
pi = 2.71; // Error
final File file = new File("readme.txt");
file = new File("LOTR.mp4"); // Error
```

final

Mit final deklarierte Variablen können nur **einmal** einen Wert zugewiesen bekommen.

```
final float pi = 3.141;
pi = 2.71; // Error
final File file = new File("readme.txt");
file = new File("LOTR.mp4"); // Error
```

Achtung!

Das final bezieht sich **nur** auf die Variable und **nicht** auf das Objekt! Somit kann das Objekt weiterhin verändert werden.

```
final File file = new File("readme.txt");
file.rename("LOTR.mp4");
```

Immutability

Immutability bedeutet, dass der Zustand eines Objekts nach der Erzeugung nicht verändert werden kann. Diese Einschränkung wird durch gezieltes **API-Design** erreicht.

```
final class Date {
  private final int day;
  private final int month;
  private final int year;

  public Date(int day, int month, int year){...}

  public int getDay(){return day;}
  public int getMonth(){return month;}
  public int getYear(){return year;}
}
```

Immutability

Vorteile unveränderlicher Objekte:

- befinden sich in exakt einem Zustand und daher simpel
- sind von Natur aus thread-sicher und benötigen keine Synchronisation
- können ohne Probleme unter Threads geteilt werden
- bilden Bauteile für weitere Objekte
- Inkonsistenzen im Zustand nicht möglich

Nachteil unveränderlicher Objekte:

$\rightarrow$ kostspielig, da für jeden distinkten Wert ein neues Objekt erzeugt wird

Just as it is a good practice to make all fields `private` unless they need greater visibility, it is a good practice to make all fields `final` unless they need to be mutable.

1.8 Strings

Strings

- Strings repräsentieren Zeichenketten
- Strings sind Objekte
- Strings sind **immutable**

```
String str = "I am a String!"; // String Literal
String str2 = new String("I am also a String!"); // DON'T DO THIS
String str3 = str2.replace("a", "b");
```

Strings

- Strings repräsentieren Zeichenketten
- Strings sind Objekte
- Strings sind **immutable**

```
String str = "I am a String!"; // String Literal
String str2 = new String("I am also a String!"); // DON'T DO THIS
String str3 = str2.replace("a", "b");
```

Gibt es einen Unterschied zwischen den beiden Methoden?

Strings

String Literale können **interned** werden.

```
String s1 = "I am a String!";
String s2 = "I am a String!";
System.out.println(s1 == s2); // true
System.out.println(s1.equals(s2)); // true
```

```
String s1 = new String("I am a String!");
String s2 = new String("I am a String!");
System.out.println(s1 == s2); // false
System.out.println(s1.equals(s2)); // true
```

Strings

String Literale können **interned** werden.

```
String s1 = "I am a String!";
String s2 = "I am a String!";
System.out.println(s1 == s2); // true
System.out.println(s1.equals(s2)); // true
```

```
String s1 = new String("I am a String!");
String s2 = new String("I am a String!");
System.out.println(s1 == s2); // false
System.out.println(s1.equals(s2)); // true
```

In der Regel werden **Strings** mit **equals** verglichen.

1.9 Arrays

Arrays

- Arrays sind Datentypen, die zur Speicherung mehrerer Werte des gleichen Typs verwendet werden
- Werden mit new angelegt → Arrays sind Objekte!
- Besitzen das Attribut **length**
- Mit dem **[]** Operator wird auf die Elemente zugegriffen
- Indizierung beginnt bei 0

```
int[] integers = new int[10];
integers[0] = 123;
integers[9] = 456;
System.out.println("Length = " + integers.length)
```

Arrays

- Arrays sind Datentypen, die zur Speicherung mehrerer Werte des gleichen Typs verwendet werden
- Werden mit new angelegt → Arrays sind Objekte!
- Besitzen das Attribut **length**
- Mit dem **[]** Operator wird auf die Elemente zugegriffen
- Indizierung beginnt bei 0

```
int[] integers = new int[10];
integers[0] = 123;
integers[9] = 456;
System.out.println("Length = " + integers.length)
```

Ein Array beinhaltet die gewünschte Anzahl von **Variablen** mit dem angegebenen Typ.
Achtung bei Referenzvariablen!.

```
File[] files = new File[10];
files[0].resize(123); // NullPointerException
```

1.10 main-Methode

Main Methode

- Die **main**-Methode ist der Einstiegspunkt
- Gehört zu einer **beliebigen** Klasse
- Hat keinen Rückgabewert
- Ist **public** und **statisch** → Nicht an Objekt gebunden
- Bekommt **Kommandozeilenargumente** übergeben

```
public class SomeClass {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

1.11 Schleifen

Schleifen

- Mehrere Schleifentypen
- Mit break wird die Schleife sofort beendet
- Mit continue wird direkt zum nächsten Durchlauf gesprungen

Schleifen

- Mehrere Schleifentypen
- Mit break wird die Schleife sofort beendet
- Mit continue wird direkt zum nächsten Durchlauf gesprungen

Die Verwendung von continue und break kann als Indiz für **Spaghetticode** angesehen werden

for-Schleife

```
for(Initialisierung; Bedingung; Schritt) {
  ...
}
```

```
for(int i=0; i<10; ++i) {
  System.out.println("Iteration = " + i);
}
```

while - Schleife

```
while(Bedingung) {
  ...
}
```

```
int i = 0;
while(i < 10) {
  System.out.println("Iteration = " + i);
  ++i;
}
```

do ... while - Schleife

```
do {
  ...
} while(Bedingung);
```

```
int i = 0;
do {
  ++i;
} while(i < 10);
```

Schleifen

do ... while - Schleife

```
do {
  ...
} while(Bedingung);
```

```
int i = 0;
do {
  ++i;
} while(i < 10);
```

Schleife wird mindestens **einmal** durchlaufen!

Schleifen

foreach - Schleife

```
for (Laufvariable: Array) {
  ...
}
```

```
int[] integers = new int[10];
... // Initialize integers
for (int i: integers) {
  i = 0;
}
```

```
File[] files = new File[10];
... // Initialize files
for (File file: files) {
  file.rename("");
}
```

1.12 Laufzeitkomplexität von Schleifen

$\mathcal{O}$-Notation

- Mit der $\mathcal{O}$-**Notation** wird das Wachstumsverhalten/Laufzeitkomplexität eines Algorithmus in Abhängigkeit zu einer Eingabegröße erfasst
- $\mathcal{O}$ betrachtet das **asymptotische** Verhalten (Worst Case)

$\mathcal{O}$-Notation

- Mit der $\mathcal{O}$-**Notation** wird das Wachstumsverhalten/Laufzeitkomplexität eines Algorithmus in Abhängigkeit zu einer Eingabegröße erfasst
- $\mathcal{O}$ betrachtet das **asymptotische** Verhalten (Worst Case)

```
void algorithm(int n) {
  for (int i = 0; i < n; ++i) {
    ...
  }
}
```

$\mathcal{O}$-Notation

- Mit der $\mathcal{O}$-**Notation** wird das Wachstumsverhalten/Laufzeitkomplexität eines Algorithmus in Abhängigkeit zu einer Eingabegröße erfasst
- $\mathcal{O}$ betrachtet das **asymptotische** Verhalten (Worst Case)

```
void algorithm(int n) {
  for (int i = 0; i < n; ++i) {
    ...
  }
}
```

$$\mathcal{O}(n)$$

$\mathcal{O}$-Notation

```
void algorithm(int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      ...
    }
  }
}
```

$\mathcal{O}$-Notation

```
void algorithm(int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      ...
    }
  }
}
```

$$\mathcal{O}(n^2)$$

$\mathcal{O}$-Notation

```
void algorithm(int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      ...
    }
  }
}
```

$$\mathcal{O}(n^2)$$

```
void algorithm(int n, int m) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      ...
    }
  }
}
```

$\mathcal{O}$-Notation

```
void algorithm(int n) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      ...
    }
  }
}
```

$$\mathcal{O}(n^2)$$

```
void algorithm(int n, int m) {
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m; ++j) {
      ...
    }
  }
}
```

$$\mathcal{O}(n \cdot m)$$

$\mathcal{O}$-Notation

Konstanten Summanden werden nicht berücksichtigt:

$$n + 11 = \mathcal{O}(n)$$

Nur der höchste Exponent wird berücksichtigt:

$$n^2 + n + 11 = \mathcal{O}(n^2)$$

Konstante Faktoren werden nicht berücksichtigt:

$$11n = \mathcal{O}(n)$$

Die Summe der Exponenten ist relevant:

$$n^3 \cdot 11n = \mathcal{O}(n^4)$$

$\mathcal{O}$-Notation

Bei **geschachtelten** Schleifen aufpassen!

$\mathcal{O}$-Notation

Bei **geschachtelten** Schleifen aufpassen!

Der **Worst-Case** Analyse nicht blind vertrauen und immer **konkret** messen!

1.13 Enumerations

Enumerationen

```
void doAction(int action) {
  if (action == 1) {
    moveForward();
  } else if (action == 2) {
    moveBackwards();
  }
}
```

Enumerationen

```
void doAction(int action) {
  if (action == 1) {
    moveForward();
  } else if (action == 2) {
    moveBackwards();
  }
}
```

Die Verwendung von **Magic Numbers** für Aufzählungen ist fehleranfällig und sollte vermieden werden!

Enumerationen

```
enum Action {
  MoveForwards,
  MoveBackwards
}
```

```
void doAction(Action action) {
  if (action == Action.MoveForwards) {
    moveForward();
  } else if (action == Action.MoveBackwards) {
    moveBackwards();
  }
}
```

```
Action action = Action.valueOf("MoveForwards");
String str = Action.MoveForward.toString();
```

Enumerationen

```
public enum Country {
  GERMANY("GER"), UK("GBR"), CHINA("CHN");

  private final String iso3CountryCode;

  Country(String iso3CountryCode) {
    this.iso3CountryCode = iso3CountryCode;
  }
  public String getISO3CountryCode() {
    return iso3CountryCode;
  }

  public static Country getDefault() {
    return GERMANY;
  }

  public static Country getRandom() {
    return values()[(int) (Math.random() * 3)];
  }
}
```

- ▷ *instance-controlled*
- ▷ Konstruktor implizit `private`
- ▷ Attribute, Konstruktoren und Methoden wie in einer gewöhnlichen Klasse

Fortsetzung Objektorientierung

Vier Säulen der Objektorientierung

1) **Beziehungen:**
   - Generalisierung und Spezialisierung
   - Aggregation und Komposition
2) **Datenkapselung**
3) **Vererbung**
4) **Polymorphismus**

Objektorientierung

Generalisierung

Zusammenfassen von Objekten mit ähnlichen Attributen und Methoden zu einer Klasse.

Aggregation und Komposition

Klassen können sich gegenseitig als **Attribute** enthalten.

Datenkapselung

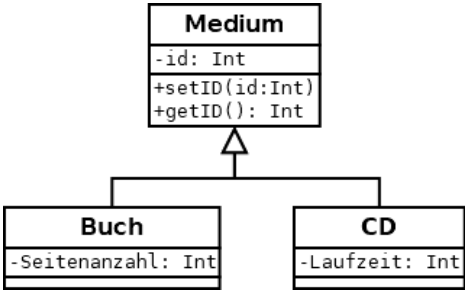
Verbergen von Implementierungsdetails durch die Verwendung von **Zugriffsmodifikatoren, Getter, Setter** und **Packages**.

1.14 Vererbung

Vererbung

Vererbung

- Ableitung einer neuen Klasse von einer bereits **vorhandenen** Klasse
- Alle Attribute und Methoden werden dabei übernommen
- Die Sub-Klasse kann erweitert werden
- Generalisierung, Spezialisierung



Vererbung

Die Vererbung wird durch das Schlüsselwort extends ausgedrückt

```
class Medium {
  private int id;
  public int getID(){...};
  public void setID(int id){...};
}
```

```
class Buch extends Medium {
  private int seitenanzahl;
  public void setSeitenanzahl(int zahl){...};
  public int getSeitenanzahl(){...};
}
```

```
Buch buch = new Buch();
buch.setID(1221);
buch.setSeitenanzahl(765);
```

Vererbung

- Jede Klasse kann nur von **einer** anderen Klasse erben
- Klassen, die von keiner anderen Klasse erben, erben **implizit** von der Klasse **Object**
- **Object** ist eine Oberklasse von jeder Klasse
- Die Vererbung drückt eine '**ist-ein**' Beziehung aus

Casting

Up-Casting

Ein Objekt einer Unterklasse kann einer Referenz einer Oberklasse zugewiesen werden.

```
Buch buch = new Buch();
Medium medium = buch; // Up-Cast
medium.setID(1221); // Ok
medium.setSeitenanzahl(765); // Error
Object object = buch; // Up-Cast
```

Casting

Up-Casting

Ein Objekt einer Unterklasse kann einer Referenz einer Oberklasse zugewiesen werden.

```
Buch buch = new Buch();
Medium medium = buch; // Up-Cast
medium.setID(1221); // Ok
medium.setSeitenanzahl(765); // Error
Object object = buch; // Up-Cast
```

Eine Variable vom Typ **Object** kann auf alle Objekte zeigen!

Casting

Down-Casting

Die Referenz einer Oberklasse kann in eine Referenz einer Unterklasse gecasted werden, solange der Typ der Objektinstanz **kompatibel** ist.

```
Medium medium = new Buch(); // Up-Cast
Buch buch = (Buch)medium; // Down-Cast
((Buch)medium).setSeitenanzahl(764); // Down-Cast
CD cd = (CD)medium; // ClassCastException
```

Casting

Down-Casting

Die Referenz einer Oberklasse kann in eine Referenz einer Unterklasse gecasted werden, solange der Typ der Objektinstanz **kompatibel** ist.

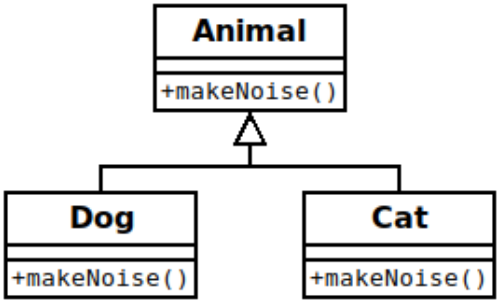
```
Medium medium = new Buch(); // Up-Cast
Buch buch = (Buch)medium; // Down-Cast
((Buch)medium).setSeitenanzahl(764); // Down-Cast
CD cd = (CD)medium; // ClassCastException
```

```
Medium medium = new Buch();
if (medium instanceof Buch) {
  var buch = (Buch)medium;
} else if (medium instanceof CD) {
  var cd = (CD)medium;
)
```

1.15.a Dynamischer Polymorphismus

Dynamischer Polymorphismus

Oftmals ist es sinnvoll, dass Methoden zwar in einer Oberklasse deklariert werden, aber sich die Implementierung je nach Unterklasse unterscheidet → **Dynamischer Polymorphismus**




```
Animal animal = new Dog();
animal.makeNoise(); // Bark!
animal = new Cat();
animal.makeNoise(); // Meow!
```

Dynamischer Polymorphismus

```
class Animal {
  public void makeNoise() { System.out.println("Generic Noise"); }
}
```

```
class Dog extends Animal {
  public void makeNoise() { System.out.println("Bark!"); }
}
```

```
class Cat extends Animal {
  public void makeNoise() { System.out.println("Meow"); }
}
```

```
Animal animal = new Dog();
animal.makeNoise(); // Bark!
animal = new Cat();
animal.makeNoise(); // Meow!
```

Dynamischer Polymorphismus

Um Fehler vorzubeugen, sollte @Override verwendet werden

```
class Animal {
  public void makeNoise() {
    System.out.println("Generic Noise");
  }
}
```

```
class Dog extends Animal {
  @Override
  public void makeNoise() {
    System.out.println("Bark!");
  }
}
```

Annotations

- Bei @Override handelt es sich um eine **Annotation**
- Mit Hilfe von **Annotationen** können Metainformationen in den Quellcode eingefügt werden
- Annotationen werden zur **Compilezeit** ausgewertet
- Es können Compilerfehler oder **zusätzlicher** Quellcode generiert werden
- Es können eigene **Annotation-Prozessoren** implementiert werden

Abstract

Abstrakte Methoden

Für manche Methoden gibt es kein **sinnvolles** Standardverhalten in der Oberklasse. Solche Methoden können als abstract markiert werden und besitzen **nur** in den Sub-Klassen eine Implementierung.

```
abstract class Animal {
  public abstract void makeNoise();
}
```

```
class Dog extends Animal {
  public void makeNoise() {
    System.out.println("Bark!");
  }
}
```

Abstract

- Klassen, die mindestens eine abstrakte Methode beinhalten, müssen als abstrakt gekennzeichnet werden
- Abstrakte Klassen können **nicht** instanziiert werden
- Jede Sub-Klasse **muss** alle abstrakten Methoden der Oberklasse implementieren oder selbst abstrakt sein
- Eine abstrakte Klasse kann weitere nicht abstrakte Methoden sowie Attribute beinhalten

Abstract

- Klassen, die mindestens eine abstrakte Methode beinhalten, müssen als abstrakt gekennzeichnet werden
- Abstrakte Klassen können **nicht** instanziiert werden
- Jede Sub-Klasse **muss** alle abstrakten Methoden der Oberklasse implementieren oder selbst abstrakt sein
- Eine abstrakte Klasse kann weitere nicht abstrakte Methoden sowie Attribute beinhalten

Mit abstrakten Klassen kann man **Schnittstellen** definieren.

Abstract

- Klassen, die mindestens eine abstrakte Methode beinhalten, müssen als abstrakt gekennzeichnet werden
- Abstrakte Klassen können **nicht** instanziiert werden
- Jede Sub-Klasse **muss** alle abstrakten Methoden der Oberklasse implementieren oder selbst abstrakt sein
- Eine abstrakte Klasse kann weitere nicht abstrakte Methoden sowie Attribute beinhalten

Mit abstrakten Klassen kann man **Schnittstellen** definieren.

Jede Klasse kann nur von **einer** Oberklasse erben ...

Interfaces

Im objektorientierten Design wollen wir das Was vom Wie trennen. Abstrakte Methoden und Interfaces sagen etwas über das Was aus, erst die konkreten Implementierungen realisieren das Wie.

Ein Interface kann enthalten:

- Konstanten (statische finale Variablen)
- abstrakte Methoden
- private und öffentliche konkrete Methoden
- private und öffentliche statische Methoden

```
public interface Buyable {
  int MAX_PRICE = 10;

  double price();

  default boolean hasPrice() {
    return price() > 0;
  }

  static boolean isValidPrice(double price) {
    return price >= 0 && price < MAX_PRICE;
  }
}
```

Interfaces

Interfaces stellen oft einen Vertrag dar. Klassen, die ein solches Interface implementiert, sollte sich an den Vertrag halten.

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

Vorgaben des Comparable-Interfaces:

- `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`
- `(x.compareTo(y) > 0 && y.compareTo(z) > 0 )` $\Longrightarrow$ x.compareTo(z) > 0
- `x.compareTo(y) == 0` $\Longrightarrow$ `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`

Interfaces

```
public final class PhoneNumber implements Comparable<PhoneNumber> {
  // ...
  @Override
  public int compareTo(PhoneNumber pn) {
    int result = Integer.compare(areaCode, pn.areaCode);
    if(result == 0) {
      result = Integer.compare(prefix, pn.prefix);
      if (result == 0) {
        result = Integer.compare(lineNum, pn.lineNum);
      }
    }
    return result;
  }
```

```
    PhoneNumber pn1 = new PhoneNumber(49,203,123456);
    PhoneNumber pn2 = new PhoneNumber(49,203,654321);
    PhoneNumber pn3 = new PhoneNumber(49,203,123654);
    // sorted internal representation for Comparable objects
    Set<PhoneNumber> phoneBook = new TreeSet<>();
    Collections.addAll(phoneBook, pn1, pn2, pn3); // pn1, pn3, pn2
```

Dynamischer Polymorphismus

Der Dynamische Polymorphismus wird intensiv verwendet.

```
List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("World");

list = new LinkedList<String>();
list.add("Hello");
list.add("World");
```

Dynamischer Polymorphismus

Der Dynamische Polymorphismus wird intensiv verwendet.

```
List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("World");

list = new LinkedList<String>();
list.add("Hello");
list.add("World");
```

Aufgepasst bei der Verwendung von var!

```
var list = new ArrayList<String>();
```

1.15.b Statischer Polymorphismus

Statischer Polymorphismus

Methoden dürfen den gleichen Namen haben, solange sich die Parametertypen unterscheiden
→ **Überladung**

```
public class PrintStream extends FileOutputStream implements Appendable, Closeable {
  ...
  public void print(boolean b) { write(String.valueOf(b)); }
  public void print(char c) { write(String.valueOf(c)); }
  public void print(int i) { write(String.valueOf(i)); }
  public void print(long l) { write(String.valueOf(l)); }
  public void print(float f) { write(String.valueOf(f)); }
  public void print(String s) { write(String.valueOf(s)); }
  public void print(Object obj) { write(String.valueOf(obj)); }
  ...
}
```

1.16 Exceptions

Exceptions

Zur Fehlerbehandlung in Java werden **Exceptions** eingesetzt.

Dies ist eine Designentscheidung und sollte respektiert werden.
When in Rome ...

Exceptions

Zur Fehlerbehandlung in Java werden **Exceptions** eingesetzt.

Dies ist eine Designentscheidung und sollte respektiert werden.
When in Rome ...

- Exceptions sind Objekte
- Alle Exceptions erben von der Basisklasse Java.lang.Throwable
- Es gibt für (fast) jeden Fehlertyp eine eigene Exception
- Exceptions werden durch Vererbungshierarchien abgebildet $\mapsto$ Oberklasse Exception
- werden mit throw geworfen

```
throw new NumberFormatException();
```

Exceptions

Exceptions können mit try ... catch gefangen werden

```
try {
  mightThrowException();
} catch (NumberFormatException e) {
  // Specific Error Handling
} catch (Exception e) {
  // Generic Error Handling
}
```

Exceptions

Exceptions können mit try ... catch gefangen werden

```
try {
  mightThrowException();
} catch (NumberFormatException e) {
  // Specific Error Handling
} catch (Exception e) {
  // Generic Error Handling
}
```

Exceptions können auch aus catch-Blöcken geworfen werden.

Exceptions

finally-Blöcke werden immer ausgeführt

```
try {
  mightThrowException();
} catch (FileNotFoundException e) {
  // Specific Error Handling
} catch (Exception e) {
  // Generic Error Handling
} finally {
  closeResources();
}
```

Exceptions

Checked Exceptions müssen gefangen werden, entweder direkt an der Stelle des Auftretens, oder an der übergeordneten Stelle mit Bezug auf den throws Hinweis in der Methodensignatur.

```
void foo() throws IOException {
  mightThrowIOException();
}
```

- Checked Exceptions erben von der Klasse Exception
- Checked Exceptions werden vom Compiler gefordert, d.h. zur Compilezeit wird festgestellt, dass erforderliche Exception-Behandlung nicht angeführt ist
- Checked Exceptions sind Ausnahmen, die außerhalb des Programms liegen
- Unchecked Exceptions erben von der Klasse RuntimeException
- Unchecked Exceptions treten zur Laufzeit auf, und haben ihre Ursache in ungenügender Programmierung
- Unchecked Exceptions sollen behandelt, d.h. vom Programmierer vorhergesehen werden

1.17 Generics

Generische Typen

Wir wollen eine Klasse `Container` erzeugen, die etwas transportiert. Der Inhalt ist nicht bekannt und ein Basistyp, der alle möglichen Objekttypen repräsentiert, wird verwendet.

```
public class Container {
  private Object load;

  public Container() {}

  public Container(Object load) {
    this.load = load;
  }

  public void setLoad(Object load)
      {
    this.load = load;
  }

  public Object getLoad() {
    return load;
  }
}
```

```
public class Ship {
  Container container;
}
```

```
Ship ship = new Ship();
ship.container = new Container(1000000L);
Long val = (Long) ship.container.getLoad();
```

- Typ sollte bereits beim Initialisieren festgelegt werden, sodass keine unerwarteten Typen verwendet werden
- Explizites Down-Casting sollte vermieden werden, um eine `ClassCastException` zur Laufzeit zu vermeiden

Generische Typen

```
public class Container<T> {
  private T load;

  public Container() {}

  public Container(T load) {
    this.load = load;
  }

  public void setLoad(T load) {
    this.load = load;
  }

  public T getLoad() {
    return load;
  }
}
```

Namenskonvention

Typparameter sind in der Regel einzelne Großbuchstaben.

- `T` – Typ
- `E` – Element
- `K` – Key/Schlüssel
- `V` – Value/Wert

Generische Schnittstellen

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

```
public interface Set<E> extends Collection<E> {

  boolean add(E e);
  int size();
  boolean isEmpty();
  boolean contains(Object o);
  Iterator<E> iterator()
  Object[] toArray();
  <T> T[] toArray(T[] a)
  ...
}
```

Generische Methoden und Konstruktoren

Bisher:

- `class Klassenname<T> ...`
- `interface Schnittstellenname<T> ...`

Eine an der Klassen- oder Schnittstellendeklaration angegebene Typvariable kann in allen **nichtstatischen** Eigenschaften des Typs angesprochen werden.

```
public class Container<T> {
  static void foo(T t) { } // Compilerfehler
}
```

Problem:

- (statische) Methoden mit eigener Typvariable
- unterschiedliche (statische) Methoden unterschiedlicher Typvariablen

Generische Methoden und Konstruktoren

Eine Klasse kann auch ohne Generics deklariert werden, aber *generische* Methoden besitzen.
`Mofizierer <Typvariable(n)> Rückgabetyp Methodenname(Parameter) throws-Klausel`

```
public class GenericMethods<V> {
  V value;

  public static <T> T random(T m, T n) {
    return Math.random() > 0.5 ? m : n;
  }

  public static void main(String[] args) {
    String s = random("FPT", "GPT");
    System.out.println(s);
  }
}
```

Einschränkung der Typen über Bounds

Die Deklaration eines generischen Typs kann durch **extends** `Basetype` eingeschränkt werden.

- `T` kann nur noch vom Typ `CharSequence` oder seiner Subklassen sein.

```
public static <T extends CharSequence > T random(T m, T n) {
  return Math.random() > 0.5 ? m : n;
  }
public static void main(String[] args) {
  String s = random("FPT",new StringBuilder("GPT")); // OK
  int i = random(10,20); // Compilerfehler
}
```

Wildcards mit ?

Wildcards repräsentieren eine Familie von Typen. ? steht nicht für `Object`, sondern für einen unbekannten Typen. Anders als bei Typvariablen sind Instanziierungen mit Wildcards nicht erlaubt und können auch nicht in Methoden genutzt werden.

```
Container<?> c = new Container<Long>(); // OK
Container<?> c = new Container<?>(); // Compilerfehler

static <T> T random(T n, T m) {...} // OK
static <?> ? random(? m, ? n) {...} // Compilerfehler
```

```
public static boolean isOneContainerEmpty(Container<?>... containers) {
  for (Container<?> container : containers) {
    if (container.isEmpty()) {
      return true;
    }
  }
  return false;
}
```

Bounded Wildcards

Wildcard	**Bezeichnung**	**Typargument**
?	*Wildcard-Typ*	Ist beliebig.
`? extends Typ`	*Upper-bounded Wildcard-Typ*	alle Unterklassen von `Typ` und `Typ` selbst.
`? super Typ`	*Lower-bounded Wildcard-Typ*	alle Oberklassen von `Typ` und `Typ` selbst.

? extends CharSequence	**? super String**
CharSequence	String
String	CharSequence
StringBuffer	Object
StringBuilder	
...	

1.18 Kovarianz und Kontravarianz

Kovarianz und Kontravarianz

Kovarianz

Subklassen werden akzeptiert

Kontravarianz

Superklassen werden akzeptiert

Bivarianz

Superklassen und Subklassen werden akzeptiert

Invarianz

Weder Superklassen noch Subklassen werden akzeptiert

Kovarianz und Kontravarianz

Einfache Referenzvariablen-Zuweisungen sind kovariant

```
Animal animal = new Cat(); // Ok
Cat cat = new Animal(); // Error
```

Subklassen können den Platz von Oberklassen einnehmen

Kovarianz und Kontravarianz

Arrays sind kovariant bezüglich des Elementtyps

```
Cat[] cats = {new Cat(), new Cat()};
Animal[] animals = cats; // Ok
animals[0] = new Dog(); // Error – ArrayStoreException
```

Kovarianz und Kontravarianz

Arrays sind kovariant bezüglich des Elementtyps

```
Cat[] cats = {new Cat(), new Cat()};
Animal[] animals = cats; // Ok
animals[0] = new Dog(); // Error – ArrayStoreException
```

Nützlich für generische Algorithmen und Datenstrukturen

```
void shuffle(Object[] objects);
```

Kovarianz und Kontravarianz

Rückgabewerte von überladenen Methoden sind kovariant

```
interface AnimalProducer {
  Animal[] produce(int count);
}
```

```
interface CatProducer extends AnimalProducer {
  @Override
  Cat[] produce(int count);
}
```

Kovarianz und Kontravarianz

Rückgabewerte von überladenen Methoden sind kovariant

```
interface AnimalProducer {
  Animal[] produce(int count);
}
```

```
interface CatProducer extends AnimalProducer {
  @Override
  Cat[] produce(int count);
}
```

Parameter von Methoden sind invariant

Kovarianz und Kontravarianz

Generische Klassen sind kovariant

```
List<Cat> list = new ArrayList<Cat>(); // Ok
```

Kovarianz und Kontravarianz

Generische Klassen sind kovariant

```
List<Cat> list = new ArrayList<Cat>(); // Ok
```

Generische Typen sind invariant

```
List<Animal> list = new ArrayList<Cat>(); // Error
```

Kovarianz und Kontravarianz

Generische Typen in Verwendung mit **extends bounds** sind kovariant

```
List<? extends Animal> list = new ArrayList<Cat>(); // Ok
Animal animal = list.get(0); // OK
list.add(new Animal()); // ERROR
```

```
public static double sumOfList(List<? extends Number> list) {
  double s = 0.0;
  for (Number n : list) {
    s += n.doubleValue();
  }
  return s;
}
```

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
sumOfList(li); // OK
sumOfList(ld); // OK
```

Kovarianz und Kontravarianz

Generische Typen in Verwendung mit **super bounds** sind kontravariant

```
List<? super Dog> list = new ArrayList<Dog>();
list.add(new Dog()); // OK
list.add(new GoldenRetriever()); // OK
Dog dog = list.get(0) // ERROR, EXPECTED OBJECT
```

```
public static void addNumbers(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++) {
    list.add(i);
  }
}
```

```
List<Integer> listOfIntegers = new ArrayList<>();
List<Number> listOfNumbers = new ArrayList<>();
addNumbers(listOfIntegers); // OK
addNumbers(listOfNumbers); // OK
```

1.19 Anonyme Klassen und Lambda Ausdrücke

Hinweis: Anonyme Klassen und Lambda Ausdrücke werden später ab Kapitel 4, GUI, verwendet.

Motivation

Viele Unterklassen werden nur erstellt, um eine **einzige** Methode zu implementieren oder zu überladen ...

```
class Song {
  int size;
}
```

```
List<Song> songs = Arrays.asList(...);
Collections.sort(songs, new SongComparator());
```

```
class SongComparator implements Comparator<Song> {
  @Override
  public int compare(Song o1, Song o2) {
    return (o1.size < o2.size) ? -1 : ((o1.size == o2.size) ? 0 : 1);
  }
}
```

Motivation

Viele Unterklassen werden nur erstellt, um eine **einzige** Methode zu implementieren oder zu überladen ...

```
class Song {
  int size;
}
```

```
List<Song> songs = Arrays.asList(...);
Collections.sort(songs, new SongComparator());
```

```
class SongComparator implements Comparator<Song> {
  @Override
  public int compare(Song o1, Song o2) {
    return (o1.size < o2.size) ? -1 : ((o1.size == o2.size) ? 0 : 1);
  }
}
```

Viel Schreibarbeit, Dateien und schwere Namensfindung ...

Anonyme Klasse

Anonyme Klassen:

- Werden **lokal** erstellt → keine eigene Datei oder Namen
- Haben nur eine **einzige** Instanz
- Können **mehrere** Methoden überladen

```
Comparator<Song> comp = new Comparator<Song>() {
  @Override
  public int compare(Song o1, Song o2) {
    return (o1.size < o2.size) ? -1 : ((o1.size == o2.size) ? 0 : 1);
  }
};
```

```
Collections.sort(songs, comp);
```

Anonyme Klasse

Anonyme Klassen:

- Werden **lokal** erstellt → keine eigene Datei oder Namen
- Haben nur eine **einzige** Instanz
- Können **mehrere** Methoden überladen

```
Comparator<Song> comp = new Comparator<Song>() {
  @Override
  public int compare(Song o1, Song o2) {
    return (o1.size < o2.size) ? -1 : ((o1.size == o2.size) ? 0 : 1);
  }
};
```

```
Collections.sort(songs, comp);
```

Viel **unnötiger** Quellcode wenn nur eine Methode genutzt wird ...

Funktionale Interfaces

Funktionale Interfaces:

- Werden mit dem interface Schlüsselwort definiert
- Besitzen nur **eine** abstrakte Methode
- **Sollten** mit @FunctionalInterface annotiert werden
- Dürfen beliebig viele Methoden mit Defaultimplementierung beinhalten

```
@FunctionalInterface
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Es gibt nur eine Methode ohne Implementierung

Lambda Ausdrücke

- Funktionieren ähnlich wie anonyme Klassen
- Implementieren (die) **eine** abstrakte Methode

```
Comparator<Song> comp = (Song o1, Song o2) -> {
  return (o1.size < o2.size) ? -1 : ((o1.size == o2.size) ? 0 : 1);
};
```

Vieles kann vom Compiler **deduziert** werden

```
Comparator<Song> comp =
  (o1, o2) -> (o1.size < o2.size) ? -1 : ((o1.size == o2.size) ? 0 : 1);
```

Lambda Ausdrücke funktionieren **nicht** mit abstrakten Klassen!

Variable Capturing

Innerhalb von **Anonymen Klassen** oder **Lambda Ausdrücken** kann auch auf Variablen des umgebenden Scopes zugegriffen werden, solange diese als **effektiv final** angesehen werden können.

```
@FunctionalInterface
public interface MyFunctionalInterface {
  void myMethod();
}
```

```
// Since object is not changed, it is effectively final
var object = new SomeClass();
MyFunctionalInterface lambda = () -> {
  // Access object inside the lambda expression
  object.doAction();
};
```

1.20 Reflection

Hinweis: Reflection wird später im Unterkapitel 6.3, im Package OpenJPA_example, verwendet.

Reflection

Definition von Reflection:

Reflection (oder **Introspektion**) bedeutet, dass ein Programm seine eigene Struktur kennt und diese eventuell modifizieren kann.

Es gibt mehrere Arten von Reflection:

- **Compile Time** Reflection.
- **Dynamische** Reflection: Zur **Laufzeit** stehen Informationen über Klassen und Objekte zur Verfügung.

Dynamische Reflection wird z.B. beim Casten von Objekten verwendet.
Falsches Casten von Objekten ⇒ ClassCastException!

Reflection

Class Klasse:

- Die Klasse `Class` gehört zur **Reflection API** und stellt Methoden zur Lieferung von Metadaten über Klassen und Interfaces zur Verfügung.
- Nur die JVM kann `Class`-Objekte erzeugen, da der Konstruktor private ist, außerdem sind die Objekte unveränderbar.

Primary Expression

```
Class<?> c1 = Data.class;
```

Objekt-Methode

```
Class<?> c2 = new Data().getClass();
```

Statische Factory Methode

```
Class<?> c3 = Class.forName("Data");
```

Reflection

Über `Class` -Objekte können Informationen über **Methoden**, **Attribute** und **Constructoren** angefordert werden:

- `Constructor<?>[] getConstructors()`
- `Method[] getMethods()`
- `Field[] getFields()`

```
var classObject = Class.forName("Data");
var constructor = classObject.getConstructor();
var data = constructor.newInstance();
```

```
var classObject = Data.class;
var constructor = classObject.getConstructor(String.class);
var data = constructor.newInstance("Useful String");
```

2. Vergleichen, Kopieren, Serialisieren

Universität Duisburg-Essen
Ingenieurwissenschaften
Informatik und Angewandte Kognitionswissenschaft
Intelligente Systeme
Josef Pauli

2.1 Vergleichen von Objekten - equals()

Vergleichen

- Mit `equals()` werden zwei Objekte miteinander verglichen
- `equals()` wird von Klasse `Object` vererbt
- Die Standardimplementierung verwendet ==

```
public class Object{
  public boolean equals(Object obj) {
    return (this == obj);
  }
}
```

- Auf jedem Objekt ist `equals()` aufrufbar
- Der Entwickler muss das Verhalten **sinnvoll** implementieren
- An den equals-contract halten!

Vergleichen

Value-Typen

- Gleichheit von zwei Objekten, wenn alle Attributwerte gleich
- Auch Structural Equality genannt, die Werte (ggf. zusammengesetzt) sind das Wesentliche
- z.B. Point, Date, Integer, String ...
- Haben eher kurze Lebensdauer

Entity-Typen

- Objekte haben ein Attribut Id
- Gleichheit zweier Objekte, wenn die Id Werte gleich sind, auch Identifier Equality genannt
- Haben lange Lebensdauer und werden (oft) in eine Datei (Datenbank) serialisiert
- Attribute können sich ändern, aber es bleibt das *gleiche* Objekt (z.B. Alter einer Person ändert sich)
- Attributwerte werden nicht als primär wichtig betrachtet

Vergleichen

Wann ist eine `equals`-Methode nicht notwendig?

- Jede Instanz der Klasse ist von Natur aus einzigartig
- Es besteht keine Notwendigkeit für die Klasse Instanzen auf logische Gleichheit zu überprüfen
- Eine Oberklasse hat die `equals()`-Methode bereits überschrieben und das Verhalten der Oberklasse ist für diese Klasse angemessen
- Die Klasse ist `private` oder `package-private` und es sicher, dass die `equals()`-Methode niemals aufgerufen wird

Vergleichen

Der Equals-Contract

- *Reflexive*: Für jeden nicht-Null Referenzwert `x` muss `x.equals(x)` `true` zurückgeben
- *Symmetric*: Für jeden nicht-Null Referenzwert `x` und `y` gibt `x.equals(y)` `true` nur dann zurück, wenn `y.equals(x)` `true` zurückgibt
- *Transitive*: Für jeden nicht-Null Referenzwert `x,y,z`, falls `x.equals(y)` `true` zurückgibt und `y.equals(z)` `true` zurückgibt, dann muss `x.equals(z)` `true` zurückgeben
- *Consistent*: For jeden nicht-Null Referenzwert `x` und `y` müssen mehrmalige Aufrufe von `x.equals(y)` konsistent `true` oder konsistent `false` zurückgeben, vorausgesetzt, dass keiner der verglichenen Werte verändert wurde
- Für jeden nicht-Null Referenzwert `x` muss x.equals(null) false zurückgeben

Vergleichen

```
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Point)) return false;
    Point point = (Point) o;
    return ((x == point.x) && (y == point.y));
  }
}
```

- Vergleiche die Referenzen mittels == zur Leistungsoptimierung
- Verwende den `instanceof`-Operator, um den Typen des Arguments zu überprüfen
- Führe einen Down-Cast des Arguments zum korrekten Typen durch
- Vergleiche alle signifikanten Felder auf Gleichheit

Vergleichen

Wichtig!

Überschreibe die `hashCode()`-Methode in jeder Klasse, die `equals()` überschreibt.

Der Hash-Code-Contract

- Wird die `hashCode()`-Methode auf einem Objekt während der Ausführung einer Anwendung wiederholt aufgerufen, muss konsistent derselbe Wert zurückgegeben werden, vorausgesetzt, dass keiner der verglichenen Werte verändert wurde.
- Sind zwei Objekte nach `equals(Object)` identisch, muss der Aufruf von `hashCode()` auf beiden Objekten dasselbe Ergebnis zurückgeben.
- Sind zwei Objekte nach `equals(Object)` **ungleich**, ist es **nicht** erforderlich, dass der Aufruf von `hashCode()` auf beiden Objekten unterschiedliche Ergebnisse liefert. Distinkte Ergebnisse für unterschiedliche Objekte erhöhen jedoch die Leistung von Hash-Tabellen.

Vergleichen

```
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {...}

  @Override
  public int hashCode() {
    int result = Integer.hashCode(x);
    result = 31 * result + Integer.hashCode(y)
    return result;
  }
}
```

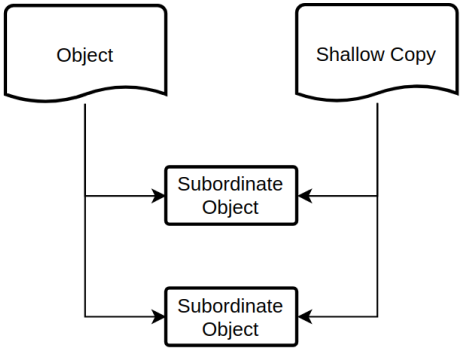
- Deklariere eine Variable `int result` und initialisiere sie mit dem Hash-Code `c` für das erste signifikante Feld
- Handelt es sich um einen primitiven Datentypen, verwende zur Berechnung `Type.hashCode(f)`, wobei `Type` die *boxed* primitive Klasse von `f` ist
- Handelt es sich um eine Objektreferenz, rufe die `hashCode()`-Methode des Objekts auf
- Kombiniere jedes signifikante Feld `c` durch die Berechnung `result = 31 * result + c`

2.2 Kopieren von Objekten - Einführung

Kopien

Flache Kopie (Shallow Copy)

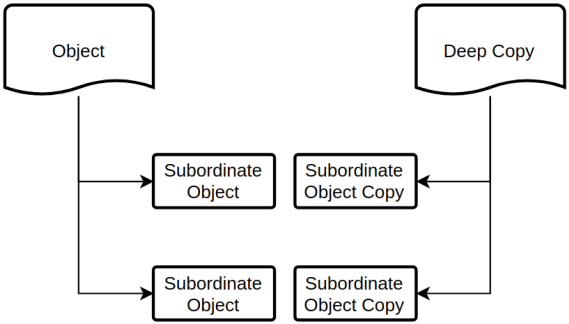
- Ein neues Objekt wird erstellt
- Alle **Variablen** werden kopiert
- Untergeordnete Objekte werden **nicht** kopiert



Kopien

Tiefe Kopie (Deep Copy)

- Ein neues Objekt wird erstellt
- Untergeordnete Objekte werden ebenfalls kopiert



Kopien

Kopien

Es gibt viele Möglichkeiten, um Kopien von Objekten zu erstellen:

- **clone()** Methode
- Copy-Constructor
- Serialisierung
- Factory-Methoden → Kapitel Entwurfsmuster
- ...

Kopien

Kopien

Es gibt viele Möglichkeiten, um Kopien von Objekten zu erstellen:

- **clone()** Methode
- Copy-Constructor
- Serialisierung
- Factory-Methoden → Kapitel Entwurfsmuster
- ...

Es wird ein **Überblick** vorgestellt und es sollte eine **Sensibilisierung** für die Schwierigkeiten und Fallstricke erfolgen ...

2.3 Kopieren von Objekten - clone()

clone

- Die Klasse **Object** besitzt die protected Methode **clone()**
- Ist native implementiert und nicht in Java
- Erstellt eine binäre Kopie eines Objektes → Flache Kopie
- Nur **Cloneable** Klassen dürfen geklont werden
  - CloneNotSupportedException
- An den clone-contract halten!

```
public class MyClass implements Cloneable {
  @Override
  public MyClass clone() {
    try {
        return (MyClass) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError(); // Can't happen
    }
  }
}
```

clone

Cloneable ist ein Marker-Interface

```
var list = new ArrayList<Object>();
...
var copy = new ArrayList<Object>();
for (var object : list) {
  if (object instanceof Cloneable) {
    copy.add(((Cloneable)object).clone()); // Error!
  }
}
```

clone() is broken ...

clone() muss gegebenenfalls über Reflections aufgerufen werden

2.4 Copy-Constructor

Copy-Constructor

- Der **Copy-Constructor** nimmt nur einen Parameter an
- Der Parameter hat als Typen **dieselbe** Klasse
- Der **Copy-Constructor** initialisiert eine Kopie
- Viele Klassen bieten einen **Copy-Constructor** an

```
class MyClass {
  private List<Animal> list;

  public MyClass(MyClass other) { // Copy Constructor
    list = new ArrayList(other.list); // Shallow-Copy!
  }
}
```

Flache und Tiefe-Kopien beachten!

2.5 try-with-resources

try-with-resources Statement

Automatische Speicherbereinigung erkennt nicht mehr referenzierte Objekte und gibt ihren Speicher frei → gilt nicht für Ressourcen

- Dateisystemressourcen von Dateien
- Netzwerkressourcen wie Socket-Verbindungen
- Datenbankverbindungen
- Nativ gebundene Ressourcen des Grafiksubsystems
- Synchronisationsobjekte

- Manche Objekte **müssen** oder **sollten** korrekt **beendet** oder **geschlossen** werden
  - z.B. durch Aufruf einer **close()** Methode
- Dies geschieht oft manuell über den **finally** Block

try-with-resources Statement

Try-catch-finally gibt Ressourcen wieder frei, jedoch mit viel Quellcode und drei Unfeinheiten:

- Soll eine Variable in finally zugänglich sein, muss sie außerhalb des try-Blocks deklariert werden, was ihr eine längere Sichtbarkeit als nötig gibt
- Das Schließen einer Ressource bringt oft ein zusätzliches try-catch mit sich
- Eine im finally ausgelöste Exception (z.B. beim `close()`) überdeckt die im try-Block ausgelöste Exception

try-with-resources Statement

```
FileOutputStream fos = null;
ObjectOutputStream oos = null;
try {
  fos = new FileOutputStream("d.ser");
  oos = new ObjectOutputStream(fos);
  oos.writeObject(myObject);
} catch (IOException e) { e.printStackTrace();}
finally {
  if(oos != null) {
    try {
      oos.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  if(fos != null) {
    //Analog zum ObjectOutputStream
  }
}
```

try-with-resources Statement

- Beim **try-with-resources** Statement wird **close()** automatisch aufgerufen
- Alle Objekte, die das korrekte Interface implementieren, können als **try-with-resources** verwendet werden
  - **java.lang.AutoCloseable**
  - **java.io.Closeable**
- Nützliche Links:
  - **try-with-resources** Statement I
  - **try-with-resources** Statement II
  - **try-with-resources** Statement III

try-with-resources Statement

```
try ( FileOutputStream fos = new FileOutputStream("d.ser");
  ObjectOutputStream oos = new ObjectOutputStream(fos))
{
  oos.writeObject(myObject);
  oos.flush();
} catch (IOException e) {
  e.printStackTrace();
}
```

2.6 Serialisierung - Motivation

Serialisierung

Die Serialisierung ist in der Informatik eine Abbildung von **strukturierten** Daten auf eine **sequenzielle** Darstellungsform.

Serialisierung wird hauptsächlich für die **Persistierung** von Objekten in Dateien und für die **Übertragung** von Objekten über das Netzwerk bei verteilten Softwaresystemen verwendet.

Die Umkehrung der **Serialisierung**, also die Umwandlung eines Datenstroms in Objekte, wird als **Deserialisierung** bezeichnet.

Einführung

Java bietet mehrere Möglichkeiten, um Daten zu serialisieren:

Binäre Serialisierung

- Serialisierung mit **java.io.ObjectOutputStream**
- Deserialisierung mit **java.io.ObjectInputStream**

XML Serialisierung

- Serialisierung mit **java.beans.XMLEncoder**
- Deserialisierung mit **java.beans.XMLDecoder**

Einführung

Java bietet mehrere Möglichkeiten, um Daten zu serialisieren:

Binäre Serialisierung

- Serialisierung mit **java.io.ObjectOutputStream**
- Deserialisierung mit **java.io.ObjectInputStream**

XML Serialisierung

- Serialisierung mit **java.beans.XMLEncoder**
- Deserialisierung mit **java.beans.XMLDecoder**

Viele Frameworks bieten weitere Möglichkeiten an

2.7 Binäre Serialisierung

Binäre Serialisierung

Binäre Serialisierung:

- Klassen, deren Objekte serialisiert werden sollen, müssen das Marker-Interface **Serializable** implementieren
- static Attribute werden nicht serialisiert, da sie nicht zum Objekt sondern zu der Klasse gehören
- Mit transient können explizit Attribute von der default-Serialisierung ausgeschlossen werden (z.B. Filehandles, Sockets)
- **Alle** nicht static/transient Attribute müssen serialisierbar sein

Binäre Serialisierung

Binäre Serialisierung:

- Klassen, deren Objekte serialisiert werden sollen, müssen das Marker-Interface **Serializable** implementieren
- static Attribute werden nicht serialisiert, da sie nicht zum Objekt sondern zu der Klasse gehören
- Mit transient können explizit Attribute von der default-Serialisierung ausgeschlossen werden (z.B. Filehandles, Sockets)
- **Alle** nicht static/transient Attribute müssen serialisierbar sein

- Die Serialisierbarkeit wird erst zur **Laufzeit** überpüft
- Nur Attribute != null werden versucht zu serialisieren
- Im Fehler wird **java.io.NotSerializableException** geworfen

Binäre Serialisierung

Versionierung:

- Jede Klasse besitzt eine **serialVersionUID**
  1) Wird vom Compiler automatisch generiert oder ...
  2) .. wird vom Benutzer festgelegt
- Stimmen die **serialVersionUIDs** nicht überein, so wird eine InvalidClassException geworfen
- Wird zwar vererbt, aber nicht berücksichtigt → private

Binäre Serialisierung

Versionierung:

- Jede Klasse besitzt eine **serialVersionUID**
  1) Wird vom Compiler automatisch generiert oder ...
  2) .. wird vom Benutzer festgelegt
- Stimmen die **serialVersionUIDs** nicht überein, so wird eine InvalidClassException geworfen
- Wird zwar vererbt, aber nicht berücksichtigt → private

Die **serialVersionUIDs** sollte vom Entwickler festgelegt werden, da die automatische Berechnung abhängig vom **Java-Compiler** ist!

```
class MySerializeableClass implements Serializeable {
  private static final long serialVersionUID = 1l;
}
```

Binäre Serialisierung

- Implementiert eine Oberklasse das Serializeable Interface, so implementieren es auch alle Unterklassen **implizit**
- Unterklassen können **nicht explizit** als NonSerializeable markiert werden

Das Serializeable Interface ist ein Vertrag!

```
class A implements Serializeable {
  private static final long serialVersionUID = 1L;
}
```

```
class B extends A {
  private void writeObject(ObjectOutputStream oos) {
    throw new NotSerializableException();
  }
}
```

Binäre Serialisierung

- Implementiert eine Oberklasse das Serializeable Interface, so implementieren es auch alle Unterklassen **implizit**
- Unterklassen können **nicht explizit** als NonSerializeable markiert werden

Das Serializeable Interface ist ein Vertrag!

```
class A implements Serializeable {
  private static final long serialVersionUID = 1L;
}
```

```
class B extends A {
  private void writeObject(ObjectOutputStream oos) {
    throw new NotSerializableException();
  }
}
```

Implementiert nur eine Unterklasse das Serializeable Interface, so werden Attribute der Oberklasse(n) ignoriert!

Binäre Serialisierung

Binäre Deserialisierung:

- Das deserialisierte Objekt hat denselben Zustand wie das serialisierte, mit Ausnahme von static und transient gekennzeichneten Attributen
- Beim Deserialisieren werden **neue** oder mit transient gekennzeichnete Attribute mit einem typspezifischen Standardwert belegt
- Objekte können nur deserialisiert werden, wenn der **Bytecode** der Klasse vorhanden ist. Dieser wird **nicht** mit in das serialisierte Objekt geschrieben

Binäre Serialisierung

Binäre Deserialisierung:

- Das deserialisierte Objekt hat denselben Zustand wie das serialisierte, mit Ausnahme von static und transient gekennzeichneten Attributen
- Beim Deserialisieren werden **neue** oder mit transient gekennzeichnete Attribute mit einem typspezifischen Standardwert belegt
- Objekte können nur deserialisiert werden, wenn der **Bytecode** der Klasse vorhanden ist. Dieser wird **nicht** mit in das serialisierte Objekt geschrieben

Auf die Kompatibilität achten!

Binäre Serialisierung

Binäre Serialisierung:

```
try (var fos = new FileOutputStream("data.ser");
     var oos = new ObjectOutputStream(fos)) {
  oos.writeObject(myObject);
} catch (Exception e) {
  ...
}
```

Binäre Deserialisierung:

```
try (var fis = new FileInputStream("data.ser");
     var ois = new ObjectInputStream(fis)) {
  var readObject = (MyObject)ois.readObject();
} catch (Exception e) {
  ...
}
```

Binäre Serialisierung

Der Serialisierungsprozess kann für bestimmte Klassen angepasst werden. Dies ist insbesondere für transiente Member nützlich.

```
private void writeObject(ObjectOutputStream oos)
        throws IOException {
  oos.defaultWriteObject(); // Default Serialization
  ... // Custom Serialization
}

private void readObject(ObjectInputStream ois)
        throws ClassNotFoundException, IOException {
  ois.defaultReadObject(); // Default Deserialization
  ... // Custom Deserialization
}
```

Die default-serialisierte Form nur dann verwenden, wenn die physische Darstellung eines Objekts identisch zu ihrem logischen Inhalt ist.

Binäre Serialisierung

```
// Geeignet für default-serialisierte Form
public class Name implements Serializable {
  private String firstName;
  private String middleName;
  private String lastName;
  ...
}
```

```
// Ungeeignet für default-serialisierte Form
public class StringList implements Serializable {
  private int size = 0;
  private Entry head = null;

  private static class Entry implements Serializable {
    String data;
    Entry next;
    Entry previous;
  }
  ...
}
```

2.8 XML Serialisierung mit Java Beans

XML Serialisierung

Was sind Java Beans?

- Java Beans sind Software-Komponenten für die Programmiersprache Java → Spezifikation (ca. 100 Seiten)
- Ein Java Bean ist eine Java Klasse, die bestimmte Bedingungen erfüllt
- Wurden im GUI-Kontext eingeführt zur einfachen Instantiierung und Übertragung von GUI-Klassen
- Erlauben automatisierten Zugriff auf ihre Eigenschaften und Methoden → siehe später Reflections

Anforderungen an einen Java Bean

- Besitzt einen **parameterlosen** Constructor
- Alle Attribute sind private
- Besitzt public **Getter** und **Setter**
- Implementiert das Marker-Interface **Serializable**

Java Bean - Properties mit Getter und Setter

```
public class JavaBeanClass implements Serializable {
  private int size;
  private String name;
  private OtherBeanClass otherBean;

  public JavaBeanClass() { ... }
  public void setSize(int size) { ... }
  public void setName(String name) { ... }
  public void setOtherBean(OtherBeanClass other) { ... }

  public int getSize() { ... }
  public String getName() { ... }
  public OtherBeanClass getOtherBean() { ... }
}
```

Andere Bean Klassen können als Attribute verwendet werden

XML Serialisierung

XML Java Beans Serialisierung:

```
var object = new JavaBeanClass();
...
try (var fos = new FileOutputStream("data.xml");
     var encoder = new XMLEncoder(fos)) {
   encoder.writeObject(object);
} catch (Exception e) {
  ...
}
```

XML Java Beans Deserialisierung:

```
try (var fis = new FileInputStream("data.xml");
     var decoder = new XMLDecoder(fis)) {
    var object = (JavaBeanClass)decoder.readObject();
} catch (Exception e) {
  ...
}
```

3. Muster

Universität Duisburg-Essen
Ingenieurwissenschaften
Informatik und Angewandte Kognitionswissenschaft
Intelligente Systeme
Josef Pauli

3.1 Einführung Muster

Einführung Muster

Warum überhaupt Muster verwenden?

- Softwareentwicklung ist ein komplexer Vorgang
- **Fehler sind teuer**
- Auf **bewährte** Lösungen zurückgreifen
- Häufige Fehlerquellen vermeiden
- Vorhandenes Wissen nutzen

Welche Mustertypen gibt es?

Prozessmuster

Wie sieht der Entwicklungs**prozess** aus?

Architekturmuster

Wie ist die **grundlegende** Architektur des Software?

Entwurfsmuster

Wie wird ein **konkretes** Problem innerhalb der Software gelöst?

Prozessmuster

Prozessmuster

Beschreibt die Schritte der Software-Entwicklung und deren Reihenfolge

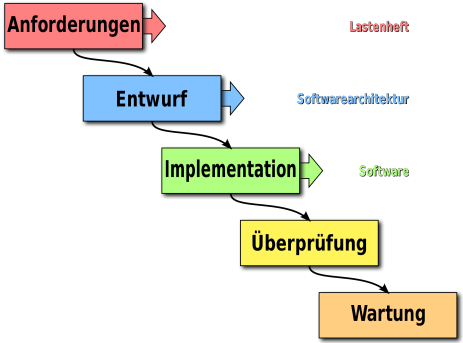


Abbildung: Wasserfallmodell

Prozessmuster

Beschreibt die Schritte der Software-Entwicklung und deren Reihenfolge

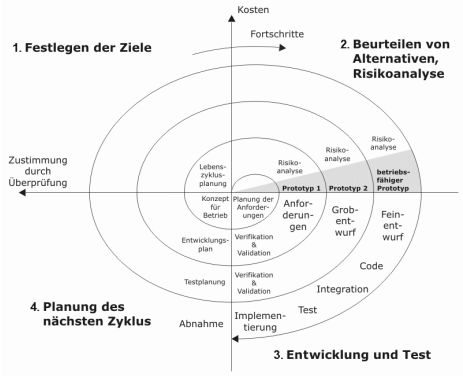


Abbildung: Spiralmodell

Prozessmuster

Es gibt eine Vielzahl von weiteren Prozessmodellen:

- Wasserfallmodell
- Spiralmodell
- Scrum
- Test Driven Development
- V-Modell (Öffentlicher Dienst)
- Extreme Programming
- weitere ...

Prozessmuster

Es gibt eine Vielzahl von weiteren Prozessmodellen:

- Wasserfallmodell
- Spiralmodell
- Scrum
- Test Driven Development
- V-Modell (Öffentlicher Dienst)
- Extreme Programming
- weitere ...

Mehr dazu in der Vorlesung **Softwaretechnik**!

Architekturmuster

Architekturmuster

Wie ist die **grundlegende** Architektur des Software?

Architekturmuster:

- Schichtenarchitektur
- Pipes und Filter
- **Model View Controller**
- **Client-Server**
- Peer-to-Peer
- ...

Entwurfsmuster

Entwurfsmuster

Achtung

Entwurfsmuster erfassen die **Essenz** eines Problems und **skizzieren** eine mögliche Lösung.
Es sind **keine** strikten Vorgehensweisen!

Es gibt eine Vielzahl von Entwurfsmustern

Erzeugungsmuster (creational patterns)

Abstraktion von Objekterzeugungsprozesse.

Strukturmuster (structural patterns)

Strukturmuster fassen Klassen und Objekte zu größeren Strukturen zusammen.

Verhaltensmuster (behavioral patterns)

Verhaltensmuster beschreiben die Interaktion zwischen Objekten und komplexen Kontrollflüssen.

Organisation der Entwurfsmuster (Gamma et al.)

		Zweck		
		Creational	**Structural**	**Behavioral**
Abstraktion	**Klasse**	Factory Method	Adapter (class)	Interpreter Template Method
	Objekt	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Abbildung: (Design Patterns. Elements of Reusable Object-Oriented Software)

3.2 Composite Pattern (Kompositum)

– Strukturmuster –

Grundlagen

- Das Kompositionsmuster wird angewendet, um Teil-Ganzes-Hierarchien zu repräsentieren, indem Objekte zu Baumstrukturen zusammengefügt werden.
- Anfragen an die Objekte sollen **aus Anwendersicht** auf gleiche Art und Weise erfolgen.
- Durch eine abstrakte Klasse oder Interface werden sowohl primitive Objekte als auch ihre Behälter repräsentiert.

Ein Beispiel sind hierarchischen Dateisystemdarstellungen, in denen einzelne Dateien und Verzeichnisse in übergeordneten Verzeichnissen organisiert sind.

Bestandteile

- **Komponente** (eng. Component)
  - Interface definiert das gemeinsame Verhalten aller Teilnehmer
- **Blatt** (eng. Leaf)
  - Definiert Verhalten von Component ohne Kinder
  - Beinhaltet konkrete Implementierung des Verhaltens
- **Kompositum** (eng. Composite)
  - Definiert das Verhalten einer Component mit Kindern
  - Implementiert die zur Verwaltung der Kinder relevanten Operationen

Composite Pattern

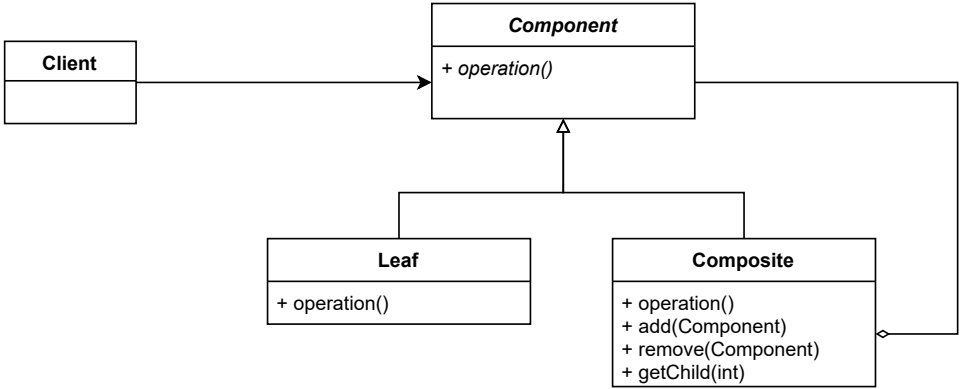


Abbildung: UML-Diagramm des Composite Patterns nach Type Safety

Composite Pattern

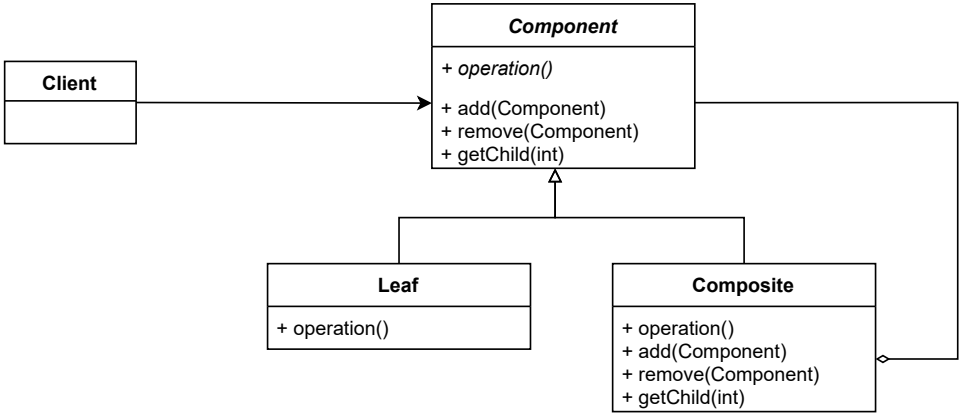


Abbildung: UML-Diagramm des Composite Patterns nach Uniformity

Composite Pattern

Behandlung von **Requests**:

- Ist der Empfänger des Requests ein **Leaf**, so wird der Request direkt behandelt, ansonsten erfolgt eine **Weiterleitung** an die Kinder.
- Der Client „weiß“ davon nichts $\rightarrow$ Vereinfachung des Client-Codes.
- Auch Hinzufügen bzw. Entfernen von **Components** erfordert keine Änderung des Client-Codes.

Composite Pattern

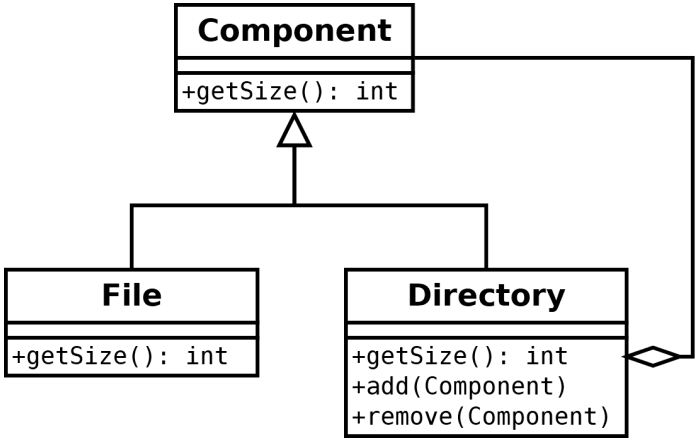


Abbildung: Spezielles UML-Diagramm nach Type Safety

Composite Pattern

Component

```
// Component
interface Component {
  int getSize();
}
```

Leaf

```
// Leaf
class File implements Component {
  public int getSize() {
    ... // Implementation
  }
}
```

Composite

```
// Composite
class Directory implements Component {
  private List<Component> components ...

  public void addComponent(Component c) { ... }
  public void removeComponent(Component c) { ... }

  public int getSize() {
    int totalSize = 0;
    for (Component c: components) {
      totalSize += c.getSize();
    }
    return totalSize;
  }
}
```

3.3 Singleton Pattern (Einzelstück)

– Erzeugungsmuster –

Singleton Pattern

Motivation

Das Singleton Muster stellt sicher, dass von einer Klasse global **maximal** eine Instanz existiert

- In der Regel ist dieses Objekt **global** verfügbar
- Mögliche Anwendungsgebiete sind der Zugriff auf Hardware oder Logger

```
Logger.getInstance().log("Important Message");
```

Singleton Pattern

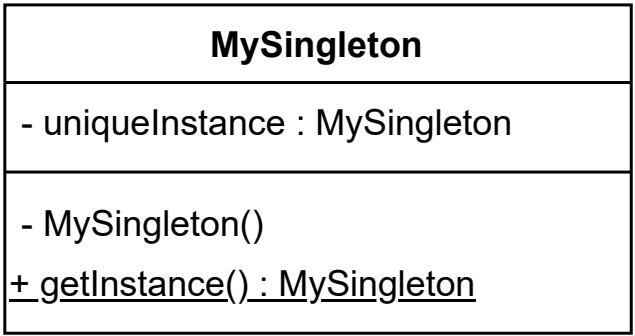


Abbildung: Klassische **Singleton** Umsetzung

Singleton Pattern

Singleton

```
public class MySingleton {
  private static MySingleton uniqueInstance = new MySingleton();

  private MySingleton() { ... }

  public static MySingleton getInstance() {
    return uniqueInstance;
  }

  public void someMethod() { ... }
}
```

Zugriff auf das Objekt

```
MySingleton.getInstance().someMethod();
```

Alternativ auch mit einem Enum

```
public enum MySingleton {
  UNIQUE_INSTANCE;

  public void someMethod() {...}

}
```

Zugriff auf das Objekt

```
MySingleton.UNIQUE_INSTANCE.someMethod();
```

Singleton Pattern

Evil Singleton

- Verstößt gegen das Single Responsibility Principle, da ein Singleton sowohl für die Geschäftslogik, als auch für die Objekterzeugung in einer Klasse zuständig ist
- Abhängigkeit von einem Singleton wird nicht aus der Schnittstelle einer Klasse oder einer Methodensignatur ersichtlich
- Extensiver Gebrauch verleitet zur prozeduralen Programmierung
- Erhöht die Kopplung
- Keine Subklassen möglich

Falsche Anwendung des Singleton Pattern

```
void doComplexCalculation(int alpha) {
  return alpha * Config.getInstance().getBeta();
}
```

3.4 Static Factory Method

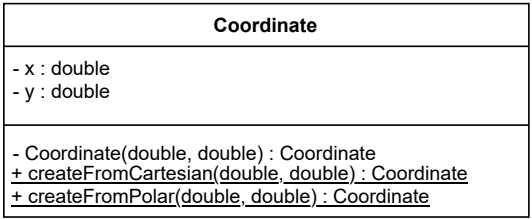
(– Erzeugungsmuster –)

Static Factory Method

Eine statische Fabrikmethode bietet eine alternative Möglichkeit der Objekterzeugung. Dabei gibt eine statische Methode als Rückgabewert eine Instanz der Klasse zurück. Eine Klasse kann Anwendern statische Fabrikmethoden anstatt oder zusätzlich zu Konstruktoren zur Verfügung stellen.

```
public final class Boolean implements ... {

  public static final Boolean TRUE = new Boolean(true);
  public static final Boolean FALSE = new Boolean(false);
  ...
  public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
  }
}
```

Static Factory Method




```
public class Coordinate {
  private double x, y;
  private Coordinate(double x, double y) {this.x = x; this.y = y;}

  public static Coordinate createFromCartesian(double x, double y) {
    return new Coordinate(x, y);}

  public static Coordinate createFromPolar(double distance, double angle) {
    return new Coordinate(distance * Math.cos(angle), distance * Math.sin(angle));}
}
```

Static Factory Method

Namensgebung: Anders als Konstruktoren, erlauben statische Fabrikmethoden eine freie Namenswahl zur besseren Lesbarkeit des Codes.

Was sind die Eigenschaften eines Objekts, das von einem Konstruktor `BigInteger(int, int, Random)` erzeugt wird?

Static Factory Method

Namensgebung: Anders als Konstruktoren, erlauben statische Fabrikmethoden eine freie Namenswahl zur besseren Lesbarkeit des Codes.

Was sind die Eigenschaften eines Objekts, das von einem Konstruktor `BigInteger(int, int, Random)` erzeugt wird?

```
/**
* Constructs a randomly generated positive BigInteger that is probably
* prime, with the specified bitLength. (...)
*/
public BigInteger(int bitLength, int certainty, Random rnd) {...}
```

Static Factory Method

Namensgebung: Anders als Konstruktoren, erlauben statische Fabrikmethoden eine freie Namenswahl zur besseren Lesbarkeit des Codes.

Was sind die Eigenschaften eines Objekts, das von einem Konstruktor `BigInteger(int, int, Random)` erzeugt wird?

```
/**
* Constructs a randomly generated positive BigInteger that is probably
* prime, with the specified bitLength. (...)
*/
public BigInteger(int bitLength, int certainty, Random rnd) {...}
```

```
/**
* Returns a positive BigInteger that is probably prime, with the
* specified bitLength. (...)
*/
public static Biginteger probablePrime(int bitLength, Random rnd) {...}
```

Static Factory Method

Objekterzeugung: Anders als Konstruktoren müssen statische Fabrikmethoden nicht bei jedem Aufruf neue Objekte erzeugen.

Static Factory Method

Objekterzeugung: Anders als Konstruktoren müssen statische Fabrikmethoden nicht bei jedem Aufruf neue Objekte erzeugen.

```
public final class Boolean implements ... {

  public static final Boolean TRUE = new Boolean(true);
  public static final Boolean FALSE = new Boolean(false);
  ...
  public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
  }
}
```

Static Factory Method

Kovarianz: Statische Fabrikmethoden können Objekte aller Subtypen ihres eigenen Rückgabetypen zurückgeben.

```
public class Collections {
  ...

  public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)
      {
    return new UnmodifiableCollection<>(c);
  }
  public static <T> List<T> unmodifiableList(List<? extends T> list) {
    return (list instanceof RandomAccess ?
      new UnmodifiableRandomAccessList<>(list) :
      new UnmodifiableList<>(list));
  }

  public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
    return new SynchronizedMap<>(m);
  }
}
```

Static Factory Method

Die Klasse eines zurückgegebenen Objekts kann sich von Aufruf zu Aufruf als eine Funktion der Parameter ändern.

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {
  Enum<?>[] universe = getUniverse(elementType);
  if (universe == null)
    throw new ClassCastException(elementType + " not an enum");

  if (universe.length <= 64)
    return new RegularEnumSet<>(elementType, universe);
  else
    return new JumboEnumSet<>(elementType, universe);
}
```

Static Factory Method

Nachteile

- Klassen ohne `public` oder `protected` Konstruktoren können keine Subklassen haben.
- Statische Fabrikmethoden sind für Programmierer schwer zu finden, da sie aus der API Dokumentation nicht wie Konstruktoren hervorgehen.

- `from` – Typumwandlungsmethode
- `of` – Aggregationsmethode
- `valueOf` – Eine ausführlichere Alternative zu `from` und `of`
- `instance` oder `getInstance` – Gibt Objekt zurück
- `create` oder `newInstance` – Gibt neu erzeugtes Objekt zurück
- `getType` – Wie `getInstance`, jedoch ist die Fabrikmethode in einer anderen Klasse.
- `newType` – Wie `newInstance`, jedoch ist die Fabrikmethode in einer anderen Klasse.
- `type` – Prägnante Alternative zu `getType`, `newType`

3.5 Factory Method

– Erzeugungsmuster –

Factory Method

Definiere eine Klassenschnittstelle zur Objekterstellung, aber lasse Unterklassen entscheiden, welche Klassen instanziiert werden. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

- Klassenbasiert
- Fabrikmethode wird durch Vererbung realisiert
- Objekterstellung und Objektgebrauch werden voneinander getrennt $\rightarrow$ erfüllt SRP
- Der Objekttyp wird durch Überladung bestimmt
- Nur ein **Produkttyp** wird erstellt
- Kann zur Erzeugung von Produktvariationen parametrisiert werden

Factory Method

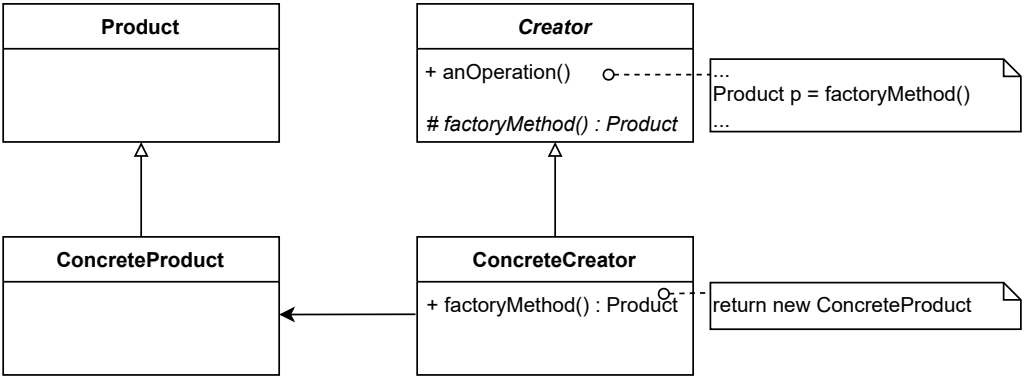


Abbildung: UML-Diagramm des Factory Method Erzeugungsmusters

3.6 Abstract Factory

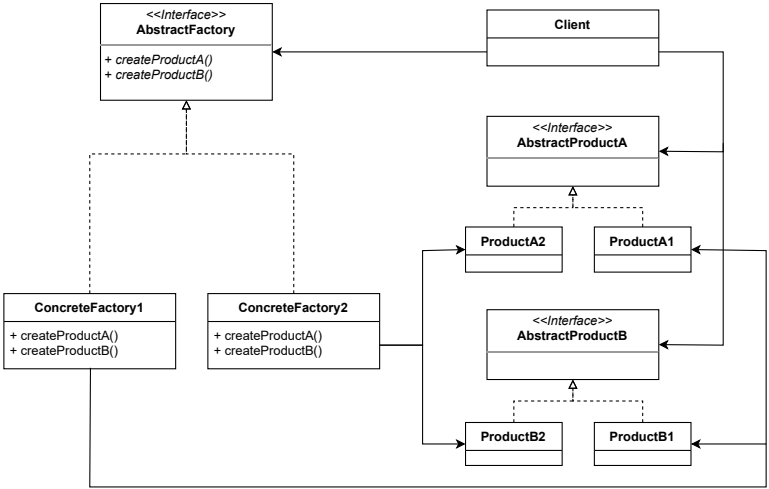
– Erzeugungsmuster –

Abstract Factory

Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

- Objektbasiert
- Komposition statt Vererbung
- Produktfamilie aus verschiedenen einzelnen Produkten wird erstellt
- Der Client programmiert gegen Abstraktionen von Produkten

Abstract Factory



3.7 Observer Pattern (Beobachter)

– Verhaltensmuster –

Observer

Motivation

Wie werden Änderungen an Objekten bekanntgemacht?

Grundlegend zwei Möglichkeiten

Push Notification

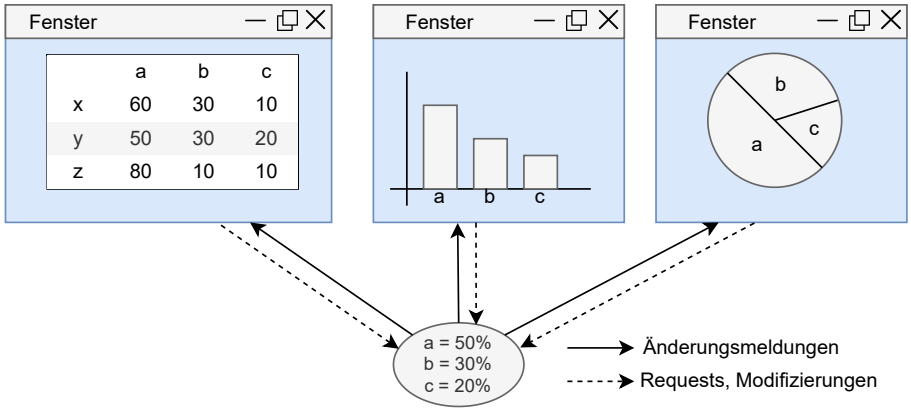
Subjekt übermittelt seinen Beobachtern ausführliche Informationen über die Zustandsänderung, ungeachtet dessen, ob sie sie haben möchten oder nicht.

Pull Notification

Subjekt übermittelt seinen Beobachtern nur die allernötigsten Informationen, z.B. dass eine Änderung stattgefunden hat. Die Beobachter erkundingen sich nach weiteren Details.

Observer

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.



Observer

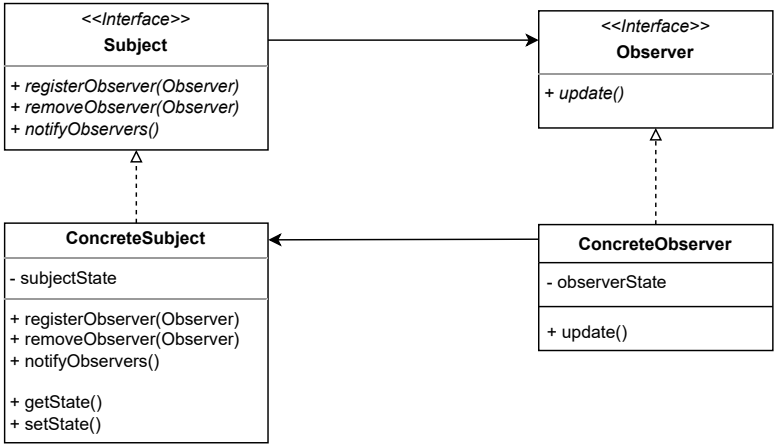


Abbildung: Klassische **Observer** Umsetzung

3.8 Anti-Pattern

Motivation für den Einsatz von Mustern

- Schlecht wartbare Software (Änderungen sind sehr aufwändig oder unmöglich)
- Längere Entwicklungszeit und höhere Kosten
- Fehlerhafte oder unsichere Implementierung

Anti-Pattern

Motivation für den Einsatz von Mustern

- Schlecht wartbare Software (Änderungen sind sehr aufwändig oder unmöglich)
- Längere Entwicklungszeit und höhere Kosten
- Fehlerhafte oder unsichere Implementierung

Schlechte Lösungen werden oft wegen **mangelnder Erfahrung** gewählt

Anti-Pattern

Muster oder Pattern sind hilfreich:

- Muster in der Software-Entwicklung und im Projektmanagement beschreiben **bewährte** Lösungsansätze für immer wiederkehrende Probleme und Aufgaben in der Software-Entwicklung → Positives Wissen wird genutzt!
- Umgekehrt sind schlechte Lösungsansätze auch lehrreich, um zukünftig Fehler zu vermeiden → **Negatives Wissen** wird genutzt!
- Daher wird das gesammelte Wissen aus Fehlern auch **Anti-Pattern** genannt.

Erschleichung von Funktionalität (Feature Creep)

- Der Umfang der im Projekt zu entwickelnden Funktionalität wird dauernd erweitert
- Dadurch können oftmals der Zeitplan und der Kostenrahmen nicht eingehalten werden
- Außerdem wird evtl. die anfangs gewählte Software-Struktur unpassend
- Manchmal wünscht sich der Auftraggeber zunächst eine geringe Funktionalität, um ein günstiges Angebot durch einen Auftragnehmer zu bekommen, und erschleicht sich später immer mehr Funktionalität zum ursprünglichen, günstigen Preis

Gegenmittel

- Möglichst genaue Spezifikation im Projektplan (Pflichtenheft)
- **Bewusste Abgrenzung von Funktionen**, die nicht umgesetzt werden sollen

Anti-Pattern

Big Ball of Mud (Große Matschkugel)

- Software, die keine erkennbare Struktur oder Software-Architektur besitzt
- Wenig aussagekräftige Funktions- und Variablennamen
- Keine/Kaum Dokumentation
- Wartbarkeit und Erweiterbarkeit sind kaum gegeben

Gegenmittel

- Nutzung von **Architekturmustern**
- Funktionalität in Pakete gliedern
- Klassen/Funktionen mit klaren Aufgabenbereichen und verständlichen Namen definieren

Anti-Pattern

Unbeabsichtigte Komplexität (Accidental Complexity)

- Es werden unnötig komplexe Lösungen für relativ einfache Probleme gewählt
- Aufwand steigt, Schritte wie Testen werden umfangreicher
- Wird auch als Over-Engineering bezeichnet

Gegenmittel

- Lösungen mit angemessener Komplexität wählen
- Detailverliebtheit einschränken

Anti-Pattern

Gott-Objekt (God object) / Gott-Klasse (God-class) / Blob

- Ein Objekt oder eine Klasse kann und/oder weiß zu viel
- Hauptklasse hat Zugriff auf viele oder sogar alle anderen Objekte
- Meist „bequem“, um schnell Ziele zu erreichen
- Aber ungünstig wartbare Struktur (keine Kapselung)
- Und schlechte Wiederverwendbarkeit des Codes, da alles mit allem anderen zusammenhängt

Gegenmittel

- Aufteilung von Verantwortlichkeiten/Funktionalitäten
- Die meisten Entwurfsmuster sorgen für bessere Kapselung

Anti-Pattern

Spaghetticode

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Fully Completed."
70 END
```

- Häufige Nutzung von Sprungbefehlen im Code
- Kontrollfluss der Software ähnelt einem Topf mit Spaghetti, Debuggen ist schwierig
- Besonders schlechte Wartbarkeit und Wiederverwendbarkeit

Gegenmittel

- Im Gegensatz zu früher, keine Sprung-Befehle nutzen!

Anti-Pattern

Noch ein aktuelles Beispiel von Spaghetticode in Java

```
int sum = 0;
int i = 0;
while (true){
  i++;
  if (i%2 == 0)
    continue;
  sum = sum + i;
  if (i>=100)
    break;
}
```

Gegenmittel

- break/continue nur in Ausnahmefällen verwenden

Anti-Pattern

Copy and Paste

```
double p1x = sin(x1*alphax*betax/gammax+obj.calc(x1,x2,x3)+x2/x1-maxx1-cos(maxx2-
    maxx3))/atan(x);
double p1y = sin(y1*alphay*betay/gammay+obj.calc(y1,y2,y3)+y2/y1-maxy1-cos(maxx2-
    maxy3))/atan(y);
```

- Quelltext wird durch Kopieren und Einfügen dupliziert und für leicht verschiedene Aufgaben modifiziert
- Aber dabei werden eventuelle Fehler mit kopiert
- Wartbarkeit ist schlecht, weil Änderungen an vielen Stellen durchgeführt werden müssen

Gegenmittel

- Niemals Quelltext-Snippets unreflektiert kopieren!!
- Ähnliche Funktionalität in Funktionen ausgliedern, durch Parameter steuern und Variablen sinnvoll benennen

Lava-Fluss (Lava-Flow)/ Dead-Code

- Ansammlung von altem und funktionslosem Code
- Entsteht meist durch Versuche oder verschiedene Versionen von Lösungen
- Statt diesen toten Code zu löschen, werden Verzweigungen um den Code herum gebaut
- Der Quelltext wird schlecht wartbar, da nicht sofort klar wird, welcher Code verwendet wird

Gegenmittel

- Toten Code zeitnah entfernen oder zumindest auskommentieren

Magische Werte (Magic Numbers)

```
double[] values = new double[42];
for(int i=7; i<23; i++){
  obj1.doSomething(values,4*i,230/10+1,0,0,0,-1);
}
```

- Es werden hart-kodierte Literale als Konstanten im Quelltext verwendet
- Verwendung von Konstanten in Funktionsaufrufen sind schlecht verständlich, besonders wenn keine Variablen-Namen oder Kommentare vorhanden sind.
- Änderung von Konstanten an verschiedenen Stellen in Quelltext ist fehleranfällig

Gegenmittel

- Konstanten immer als Variable mit aussagekräftigem Namen deklarieren und wiederverwenden

Anti-Pattern

Das Rad neu erfinden (Reinventing the Wheel)

- Stetige Neuerstellung von bereits vorhandener Software
- Bekannte Software-Bibliotheken werden nicht verwendet, auch wenn die Funktionalität bereits schneller/besser/eleganter vorhanden ist
- Macht Software-Entwicklung aufwändiger und fehleranfälliger
- Beispiel ist etwa die Implementierung eines Bubble-Sort-Algorithmus, auch wenn eine bessere Sortierfunktion bereits vorhanden ist (z.B. Collections.sort in Java)

Gegenmittel

- Nutzung von verfügbaren Bibliotheken
- Aber Software-Lizenz der Bibliotheken beachten!

Literatur und Links

- Prozessmuster:
  - Prozessmuster I
  - Prozessmuster II
  - Prozessmuster III
  - Prozessmuster IV
- Architekturmuster:
  - Architekturmuster I
  - Architekturmuster II
- Entwurfsmuster:
  - Entwurfsmuster I
  - Entwurfsmuster II
  - Entwurfsmuster III
  - Entwurfsmuster IV
  - Entwurfsmuster V

4. Nebenläufigkeit

Universität Duisburg-Essen
Ingenieurwissenschaften
Informatik und Angewandte Kognitionswissenschaft
Intelligente Systeme
Josef Pauli

4.1 Einführung in die Nebenläufigkeit

Einführung

Nebenläufigkeit

- Die **Nebenläufigkeit (Concurrency)** ist in der Informatik die Eigenschaft eines Systems, mehrere Berechnungen oder Anweisungen unabhängig voneinander ausführen zu können.
- Die Ausführungsreihenfolge ist dabei **egal** (nacheinander, gleichzeitig, abwechselnd).

Einführung

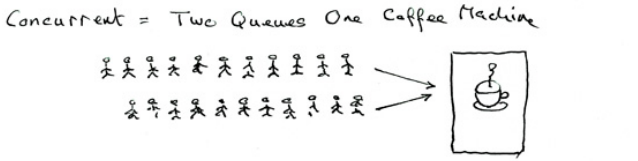
Nebenläufigkeit

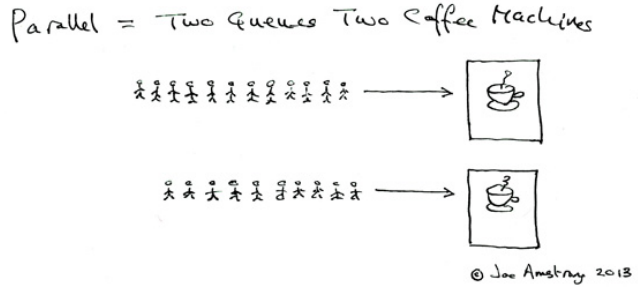
- Die **Nebenläufigkeit (Concurrency)** ist in der Informatik die Eigenschaft eines Systems, mehrere Berechnungen oder Anweisungen unabhängig voneinander ausführen zu können.
- Die Ausführungsreihenfolge ist dabei **egal** (nacheinander, gleichzeitig, abwechselnd).

Parallelität

Ein nebenläufiges System arbeitet **parallel**, wenn die Befehle **gleichzeitig** ausgeführt werden. Das System bietet somit **Parallelität (Parallelism)**.

Einführung


Concurrent = Two Queues One Coffee Machine


Parallel = Two Queues Two Coffee Machines
© Joe Armstrong 2013

Wie wird Nebenläufigkeit umgesetzt?

Wie wird Nebenläufigkeit umgesetzt?

Prozess

- Ein Prozess ist ein Programm zur Laufzeit.
- Besitzt zusätzliche Verwaltungsinformationen.
- Besitzt eigenen Speicher (Heap und Stack).

Wie wird Nebenläufigkeit umgesetzt?

Prozess

- Ein Prozess ist ein Programm zur Laufzeit.
- Besitzt zusätzliche Verwaltungsinformationen.
- Besitzt eigenen Speicher (Heap und Stack).

User-Thread (leichtgewichtiger Prozess)

- Ist einem Prozess zugeordnet.
- Teilt sich den Heap mit dem Prozess.
- Hat eigenen Stack.

Wie wird Nebenläufigkeit umgesetzt?

Prozess

- Ein Prozess ist ein Programm zur Laufzeit.
- Besitzt zusätzliche Verwaltungsinformationen.
- Besitzt eigenen Speicher (Heap und Stack).

User-Thread (leichtgewichtiger Prozess)

- Ist einem Prozess zugeordnet.
- Teilt sich den Heap mit dem Prozess.
- Hat eigenen Stack.

Umsetzung durch **Multiprocessing** oder **Multithreading**.

Umsetzung

Wie funktioniert Multithreading oder Multiprocessing mit nur einem CPU-Kern?

- Die Threads/Prozesse werden nacheinander oder abwechselnd ausgeführt.
- Dadurch behält das Programm seine **Empfänglichkeit (Responsiveness)**. Besonders für GUI-Anwendungen wichtig!
- Die Reihenfolge oder Zeitscheibeneinteilung wird von einem **Scheduler** verwaltet (Round-Robin, ...).

Umsetzung

Wie funktioniert Multithreading oder Multiprocessing mit nur einem CPU-Kern?

- Die Threads/Prozesse werden nacheinander oder abwechselnd ausgeführt.
- Dadurch behält das Programm seine **Empfänglichkeit (Responsiveness)**. Besonders für GUI-Anwendungen wichtig!
- Die Reihenfolge oder Zeitscheibeneinteilung wird von einem **Scheduler** verwaltet (Round-Robin, ...).

Die Anwendung von **Multithreading** oder **Multiprocessing** hat nicht zwangsläufig **Parallelität** zur Folge!

Threads

Verwendung von Threads:

- Threadklassen müssen entweder von Klasse **Thread** abgeleitet sein oder das Interface **Runnable** implementieren.
- Da Java keine Mehrfachvererbung unterstützt, wird oft auf das Erben von der Klasse Thread verzichtet.
- Ein Thread wird mit `start()` gestartet.
- Die Methode `run()` wird beim Starten des Threads ausgeführt.
- Ein direkter Aufruf von `run()` würde zu einer seriellen Abarbeitung führen und keine Nebenläufigkeit erzeugen.

```
interface Runnable {
  void run();
}
```

Threads mit Runnable

Beispiel einer Klasse die von Thread erbt

```
public class MyThread extends Thread {
    String greeting;
    public MyThread(String greeting) {
        this.greeting = greeting;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; ++i) {
            System.out.println(greeting);
        }
    }
}
```

```
var m1 = new MyThread("Hi");
var m2 = new MyThread("Moin");
m1.start();
m2.start();
```

Threads mit Runnable

Beispiel einer Klasse die Runnable implementiert

```
public class MyRunnable implements Runnable {
    String greeting;
    public MyRunnable(String greeting) {
        this.greeting = greeting;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; ++i) {
            System.out.println(greeting);
        }
    }
}
```

```
var m1 = new MyRunnable("Hi");
var m2 = new MyRunnable("Moin");
var t1 = new Thread(m1);
var t2 = new Thread(m2);
t1.start();
t2.start();
```

Threads mit Runnable

Starten von Threads mit Anonymen Klassen

```
// Thread 1
new Thread(new Runnable() {
  @Override
  public void run() {
    for (int i = 0; i < 10; ++i) {
      System.out.println("Hi");
    }
  }
}).start();

// Thread 2
new Thread(new Runnable() {
  @Override
  public void run() {
    for (int i = 0; i < 10; ++i) {
      System.out.println("Moin");
    }
  }
}).start();
```

Threads mit Runnable

Starten von Threads mit Lambda Ausdrücken

```
// Thread 1
new Thread(() -> {
  for (int i = 0; i < 10; ++i) {
    System.out.println("Hi");
  }
}).start();

// Thread 2
new Thread(() -> {
  for (int i = 0; i < 10; ++i) {
    System.out.println("Moin");
  }
}).start();
```

Starten von Threads mit Lambda Ausdrücken

```
// Thread 1
new Thread(() -> {
  for (int i = 0; i < 10; ++i) {
    System.out.println("Hi");
  }
}).start();

// Thread 2
new Thread(() -> {
  for (int i = 0; i < 10; ++i) {
    System.out.println("Moin");
  }
}).start();
```

Ist die Ausgabe immer gleich?

Threads mit Runnable

Daemon-Threads:

- `setDaemon()` setzt die Daemon-Eigenschaft.
- Eine Java-Applikation terminiert, wenn alle **nicht**-Daemon Threads abgearbeitet worden sind.
- Daemon-Threads sollten **keine** Hardware-Zugriffe oder IO-Aktionen ausführen.

Thread-Attribute werden auf Kinder-Threads vererbt.

Threads mit Runnable

Warten auf Threads:

- Mit der Methode `join()` kann auf die Abarbeitung von Threads gewartet werden.
- `join()` sollte erst aufgerufen werden, wenn alle Threads gestartet sind, da sonst **keine** Nebenläufigkeit mehr existiert.

```
var thread = new Thread(() -> {
  Thread.sleep(1000);
});
thread.start();
thread.join();  // Blockiert mindestens eine Sekunde
```

- Die Methode `stop()` ist deprecated! (Offizielle Begründung)

Threads sollten immer sauber beendet werden!
(später dazu mehr ...)

Threads mit Runnable

Weitere nützliche Methoden:

- `static Thread currentThread()` : Gibt das aktuelle Thread-Objekt zurück.
- `void setPriority(int)` : Setzt die Priorität (1 - 10) für die Abarbeitung:
  - `Thread.MIN_PRIORITY`
  - `Thread.NORM_PRIORITY`
  - `Thread.MAX_PRIORITY`
- `static void sleep(int ms)` stoppt die Abarbeitung des Threads für mindestens `ms` Millisekunden.
- `static void yield()` stoppt die Abarbeitung des Threads auf unbestimmte Zeit.

Thread-Attribute werden auf Kinder-Threads vererbt.

4.2 Synchronisation von Threads

Thread-Safety

Notwendigkeit für Synchronisation

Wenn mehr als ein Thread auf eine bestimmte Zustandsvariable zugreift und einer von ihnen möglicherweise darauf schreibt, müssen sie alle ihren Zugriff darauf durch Synchronisation koordinieren.

Aufgabe von threadsicherem Code

Verwaltung des Zugriffs auf den Zustand eines Objekts, insbesondere auf den gemeinsamen veränderbaren Zustand (*shared mutable state*).

- *shared*: Auf eine Variable kann von mehreren Threads zugegriffen werden.
- *mutable*: Der Wert einer Variable kann sich während seiner Lebensdauer ändern.

Thread-Safety

Threadsichere Klasse

Eine Klasse ist threadsicher, wenn sie sich korrekt verhält, wenn auf sie von mehreren Threads aus zugegriffen wird, unabhängig von der Planung oder Verschachtelung der Ausführung dieser Threads durch die Laufzeitumgebung und ohne zusätzliche Synchronisation oder andere Koordination seitens des aufrufenden Codes.

Instanz einer threadsicheren Klasse

Operationen, die sequentiell oder nebenläufig auf Instanzen einer threadsicheren Klasse ausgeführt werden, können nicht dazu führen, dass eine Instanz in einen ungültigen Zustand gerät.

Synchronisation von Threads

Nur Lesezugriff

Reiner Lesezugriff auf dieselben Daten ist unkritisch, solange **kein** Thread die Daten verändert. Dies ist allerdings selten der Fall.

Synchronisation von Threads

Nur Lesezugriff

Reiner Lesezugriff auf dieselben Daten ist unkritisch, solange **kein** Thread die Daten verändert. Dies ist allerdings selten der Fall.

Lese- und Schreibzugriff

Greifen mehrere Threads auf **dieselben** Daten zu (auch schreibend), so müssen diese Zugriffe **synchronisiert** werden, da es sonst zu inkonsistenten bzw. fehlerhaften Daten kommen kann.

Synchronisation von Threads

Nur Lesezugriff

Reiner Lesezugriff auf dieselben Daten ist unkritisch, solange **kein** Thread die Daten verändert. Dies ist allerdings selten der Fall.

Lese- und Schreibzugriff

Greifen mehrere Threads auf **dieselben** Daten zu (auch schreibend), so müssen diese Zugriffe **synchronisiert** werden, da es sonst zu inkonsistenten bzw. fehlerhaften Daten kommen kann.

Was sind Inkonsistenzen?

Synchronisation von Threads

Ohne Synchronisierung sind Inkonsistenzen möglich:

- Daten werden gleichzeitig von mehreren Threads geändert. Dies kann zu **fehlerhaften** Daten führen.
- Notwendige **Reihenfolge/Abfolge** von Befehlen wird nicht eingehalten.
- Probleme beim Ressourcen-Zugriff (insbesondere Hardware).
- Daten wurden geändert, aber die Änderungen sind in anderen Threads **nicht** oder nur **teilweise** sichtbar.

Synchronisation von Threads

Ohne Synchronisierung sind Inkonsistenzen möglich:

- Daten werden gleichzeitig von mehreren Threads geändert. Dies kann zu **fehlerhaften** Daten führen.
- Notwendige **Reihenfolge/Abfolge** von Befehlen wird nicht eingehalten.
- Probleme beim Ressourcen-Zugriff (insbesondere Hardware).
- Daten wurden geändert, aber die Änderungen sind in anderen Threads **nicht** oder nur **teilweise** sichtbar.

Probleme

- **Kritische Bereiche**, die nicht unterbrochen werden dürfen.
- **Sichtbarkeit** von Änderungen für andere Threads.

Warum sind Änderungen nicht sofort sichtbar?

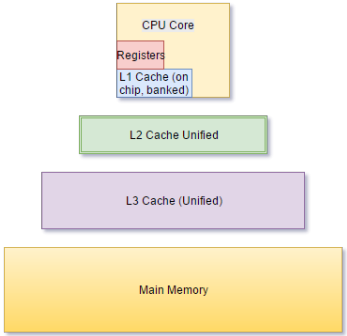


Abbildung: Vereinfachtes Speicherlayout einer CPU

Race-Condition: Read-Modify-Write

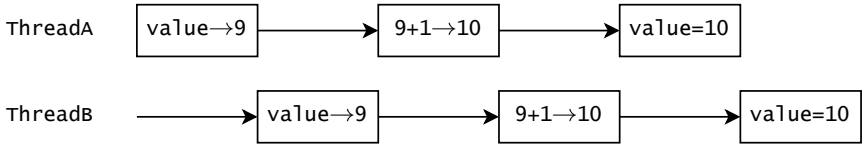


Abbildung: Beispiel einer nicht threadsicheren *read-modify-write* Operation

```
@NotThreadSafe
public class UnsafeSequence {
  private int value;

  public int getNext() {
    return value++;
  }
}
```

Race-Condition: Check-Then-Act

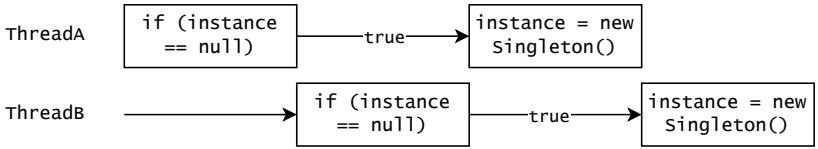


Abbildung: Beispiel einer nicht threadsicheren *check-then-act* Operation

```
@NotThreadSafe
public class Singleton {
  private static Singleton instance;
  private Singleton(){}

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();}
    return instance;
  }
}
```

Happens-Before

Happens-Before Relation:

- Java Programme werden von einer **JVM** ausgeführt. Es gibt somit eine **Abstraktionsschicht** zwischen der Anwendung und der Hardware.
- Das Verhalten von Java Applikationen ist genau spezifiziert.
- Die Spezifikation führt die **happens-before** Relation ein.

Happens-Before

Happens-Before Relation:

- Java Programme werden von einer **JVM** ausgeführt. Es gibt somit eine **Abstraktionsschicht** zwischen der Anwendung und der Hardware.
- Das Verhalten von Java Applikationen ist genau spezifiziert.
- Die Spezifikation führt die **happens-before** Relation ein.

> *Two actions can be ordered by a happens-before relationship. If one action happens-before another, then the first is visible to and ordered before the second.*

Happens-Before

Happens-Before Relation:

- Java Programme werden von einer **JVM** ausgeführt. Es gibt somit eine **Abstraktionsschicht** zwischen der Anwendung und der Hardware.
- Das Verhalten von Java Applikationen ist genau spezifiziert.
- Die Spezifikation führt die **happens-before** Relation ein.

> *Two actions can be ordered by a happens-before relationship. If one action happens-before another, then the first is visible to and ordered before the second.*

Durch Synchronisationen werden happens-before Relationen vereinbart!

Happens-Before

Happens-Before Relation:

- Java Programme werden von einer **JVM** ausgeführt. Es gibt somit eine **Abstraktionsschicht** zwischen der Anwendung und der Hardware.
- Das Verhalten von Java Applikationen ist genau spezifiziert.
- Die Spezifikation führt die **happens-before** Relation ein.

> *Two actions can be ordered by a happens-before relationship. If one action happens-before another, then the first is visible to and ordered before the second.*

Durch Synchronisationen werden happens-before Relationen vereinbart!

> *In the absence of a happens-before ordering between two operations, the JVM is free to reorder them as it pleases.*

Happens-Before

```
Thread one = new Thread(new Runnable() {
  @Override
  public void run() {
    a = 1;
    x = b;
  }
});

Thread other = new Thread(new Runnable() {
  @Override
  public void run() {
    b = 1;
    y = a;
  }
});

one.start(); other.start();
one.join(); other.join();
System.out.println("( " + x + ", " + y +" )");
```

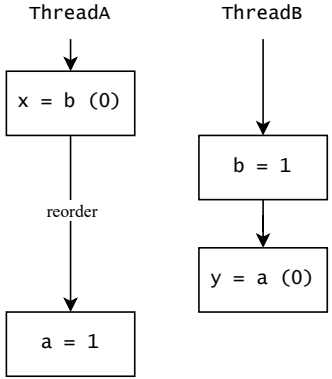


Abbildung: Neusortierung der Anweisungen durch die JVM in Abwesenheit einer *happens-before* Relation

Happens-Before

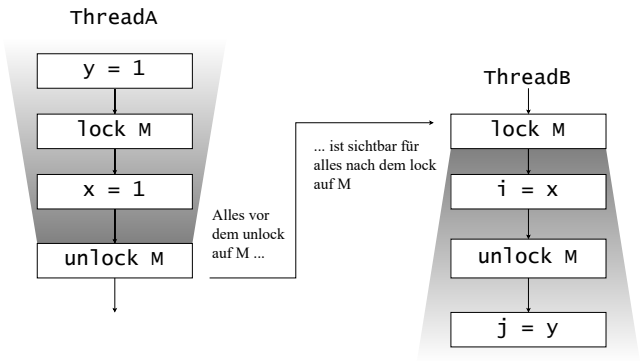


Abbildung: Visualisierung der *happens-before* Relation im Java Memory Model

Race-Condition und Data-Race

Race-Condition

Die Korrektheit einer Berechnung hängt vom relativen Timing oder der Verschachtelung mehrerer Threads durch die Laufzeitumgebung ab.

- ▷ *read-modify-write*
- ▷ *check-then-act*

Data-Race

Ein Datenwettlauf tritt auf, wenn eine Variable von mehr als einem Thread gelesen und von mindestens einem Thread geschrieben wird, aber die Lese- und Schreibvorgänge nicht nach *happens-before* geordnet sind.

Race-Condition $\neq$ **Data-Race**

Happens-Before

Race-Condition, Data Race oder sogar beides?

```
public class UnsafeLazyInitialization {
  private static Resource resource;
  public static Resource getInstance() {
    if (resource == null) {
      resource = new Resource();
    }
    return resource;
  }
}
```

Happens-Before

Race-Condition, Data Race oder sogar beides?

```
public class UnsafeLazyInitialization {
  private static Resource resource;
  public static Resource getInstance() {
    if (resource == null) {
      resource = new Resource();
    }
    return resource;
  }
}
```

- Race-Condition: *check-then-act* ohne Synchronisation
- Data-Race: keine *happens-before* Relation zwischen dem Schreiben des Objekts in einem Thread A und dem Lesen in Thread B.

Happens-Before

Race-Condition, Data Race oder sogar beides?

```
public class UnsafeLazyInitialization {
  private static Resource resource;
  public static Resource getInstance() {
    if (resource == null) {
      resource = new Resource();
    }
    return resource;
  }
}
```

- Race-Condition: *check-then-act* ohne Synchronisation
- Data-Race: keine *happens-before* Relation zwischen dem Schreiben des Objekts in einem Thread A und dem Lesen in Thread B.

> *With the exception of immutable objects, it is not safe to use an object that has been initialized by another thread unless the publication happens-before the consuming thread uses it.*

Synchronisation von Threads

Es gibt mehrere Möglichkeiten für die Synchronisation:

- Atomare Objekte (atomics)
- volatile Schlüsselwort
- synchronized Schlüsselwort
- Lock Interface

Jedes Synchronisationsverfahren hat Vor- und Nachteile!

Atomarität

Atomare Operationen

Die Operationen A und B sind in ihrer Beziehung zueinander atomar, wenn aus der Sicht eines Threads, der A ausführt, wenn ein anderer Thread B ausführt, entweder alles von B ausgeführt wurde oder nichts davon. Eine *atomare Operation* ist eine Operation, die in Bezug auf alle Operationen, einschließlich ihrer selbst, die auf demselben Zustand operieren, atomar ist.

Zusammengesetzte Operationen (*compound actions*)

- check-then-act
- read-modify-write

Volatile

- Wenn eine Variable volatile gekennzeichnet ist, wird dem Compiler und der Laufzeitumgebung mitgeteilt, dass diese Variable gemeinsam genutzt wird und dass Operationen auf ihr nicht mit anderen Speicheroperationen neu geordnet werden sollten.
- Eine volatile Variable wird nicht in Registern oder an Orten, wo sie von anderen Prozessoren verborgen sind, zwischengespeichert.
- Ein Lesezugriff auf eine volatile Variable gibt immer den zuletzt geschriebenen Wert eines beliebigen Threads zurück.
- Sperrt keine kritischen Bereiche und kann nicht dazu führen, dass der ausführende Thread blockiert.
- Keine Atomarität für *read-modify-write* Operationen, es sei denn, es kann garantiert werden, dass die Variable nur von einem einzigen Thread geschrieben wird.

Volatile

Die häufigsten Verwendungszwecke für volatile Variablen sind:

- Abschluss einer Operation
- Unterbrechung einer Operation
- Status flag

ohne die Kennzeichnung volatile im linken Code, kann JVM Hoisting durchführen

```
volatile boolean interrupted;
...
  while(!interrupted){
    doOperation();
  }
```

→

```
boolean interrupted;
...
  if (!interruped) {
    while(true) {
      doOperation();
    }
  }
```

Atomics

Atomare Datentypen:

- Unter **Atomics** versteht man eine Menge von (kleinen) Klassen, die nur synchronisierte (nicht teilbare) Zugriffe erlauben.
- In der Regel werden **Atomics** nur für Standardtypen angeboten, z.B.
  - **AtomicBoolean, AtomicInteger, AtomicLong**
- (Fast) Alle Prozessorarchitekturen unterstützen nativ **Atomics**.
- Insbesondere **getAndSet** bzw. **getAndAdd** sind wichtig!

Beispiel AtomicInteger

```
var count = new AtomicInteger();
count.set(0);

var t1 = new Thread(() -> {
  for (int i = 0; i < 10; ++i) {
    count.getAndAdd(1); // Get und Set sind synchronisiert
  }
});

var t2 = new Thread(() -> {
  for (int i = 0; i < 10; ++i) {
    count.getAndAdd(1); // Get und Set sind synchronisiert
  }
});

t1.start(); t2.start();
t1.join(); t2.join();

System.out.println(count.get()); // Gibt garantiert 20 aus!
```

Synchronized

- Intrinsiche Sperre.
- Das Schlüsselwort **synchronized** kann zum Sperren eines Blocks verwendet werden.
- Die Objektreferenz dient als Schloss.
- Das Schlüsselwort **synchronized** kann auch zum Sperren einer ganzen Methode verwendet werden.
- Dabei wird über den **this** Pointer das umgebende Objekt gesperrt oder speziell die Methode gesperrt.

```
synchronized (this) {
  // Access or modify shared state guarded by lock
}
```

```
public static synchronized Singleton getInstance() {
  // Access or modify shared state guarded by lock
}
```

Synchronized

Beispiel für die Verwendung von synchronized

```
var service = new Service();
for (int i = 0; i < 10; ++i) {
  new Thread(() -> {
    while (true) {
      synchronized(service) { // Beginn kritischer Bereich
        service.doStepOne();
        service.doStepTwo();
      } // Ende kritischer Bereich
    }
  }).start();
}
```

Synchronized

Weitere Beispiele für die Verwendung von synchronized

```
public class Service {
  public void doStep1() { ... }
  public void doStep2() { ... }
  public void doStep1And2() {
    synchronized (this) { // Beginn kritischer Bereich
      doStep1();
      doStep2();
    } // Ende kritischer Bereich
  }
}
```

```
public class Service {...
  // Beginn kritischer Bereich
  public synchronized void doStep1And2() {
    doStep1();
    doStep2();
  } // Ende kritischer Bereich
}
```

Synchronized

Vor- und Nachteile von **synchronized**:

+ Sehr einfache Handhabung.
+ Das Freigeben von kritischen Bereichen kann nicht vergessen werden.
– Kritische Bereiche können nicht über Methodengrenzen hinausgehen.
– Es ist nicht möglich, die **Fairness** zu beeinflussen.

Locks

Lock:

- Extrinsische Sperre.
- Ist eine Art von Türschloss, welches von einem Thread beim Betreten eines kritischen Bereichs zugesperrt wird.
- Befindet sich bereits ein Thread in einem kritischen Bereich, so müssen die anderen Threads warten, bis dieser den kritischen Bereich wieder frei gibt.
- Jeder Thread, der einen kritischen Bereich betritt, sieht alle Änderungen, die ein anderer Thread in diesem kritischen Bereich gemacht hat.

Beispiel für die Verwendung von Lock

```
class Service {
  void doStepOne() { ... }
  void doStepTwo() { ... }
}
```

```
Lock lock = new ReentrantLock();
var service = new Service();
for (int i = 0; i < 10; ++i) {
  new Thread(() -> {
    while (true) {
      lock.lock(); // Beginn kritischer Bereich
      service.doStepOne();
      service.doStepTwo();
      lock.unlock(); // Ende kritischer Bereich
    }
  }).start();
}
```

Beispiel für die Verwendung von Lock

```
class Service {
  void doStepOne() { ... }
  void doStepTwo() { ... }
}
```

```
Lock lock = new ReentrantLock();
var service = new Service();
for (int i = 0; i < 10; ++i) {
  new Thread(() -> {
    while (true) {
      lock.lock(); // Beginn kritischer Bereich
      service.doStepOne();
      service.doStepTwo();
      lock.unlock(); // Ende kritischer Bereich
    }
  }).start();
}
```

Bei einer Exception wird der Lock nie wieder freigegeben!

Beispiel für die Verwendung von Lock

```
Lock lock = new ReentrantLock();
var service = new Service();
for (int i = 0; i < 10; ++i) {
  new Thread(() -> {
    while (true) {
      lock.lock(); // Beginn kritischer Bereich
      try {
        service.doStepOne();
        service.doStepTwo();
      } finally {
        lock.unlock(); // Ende kritischer Bereich
      }
    }
  }).start();
}
```

Locks

Beispiel für die Verwendung von Lock

```
Lock lock = new ReentrantLock();
var service = new Service();
for (int i = 0; i < 10; ++i) {
  new Thread(() -> {
    while (true) {
      lock.lock(); // Beginn kritischer Bereich
      try {
        service.doStepOne();
        service.doStepTwo();
      } finally {
        lock.unlock(); // Ende kritischer Bereich
      }
    }
  }).start();
}
```

Bei der Verwendung von Locks ist Aufmerksamkeit geboten!

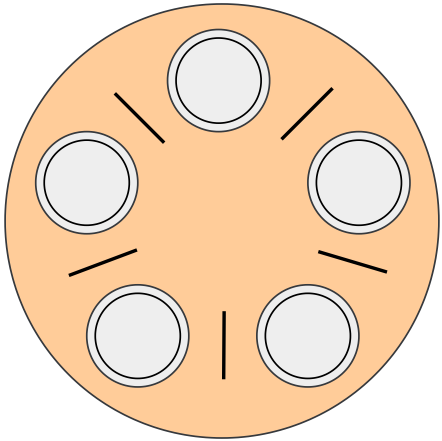
Locks

Vor- und Nachteile von **Locks**

+ Die **Fairness** kann beeinflusst werden.
+ **Kritische Bereiche können über Methodengrenzen hinausgehen.**
+ Mit **trylock** kann geprüft werden, ob ein Objekt gesperrt ist.
– Es ist nicht immer (trivial) ersichtlich, wo kritische Bereiche beginnen und enden.
– Das Freigeben von kritischen Bereichen kann vergessen werden.

4.3 Deadlocks

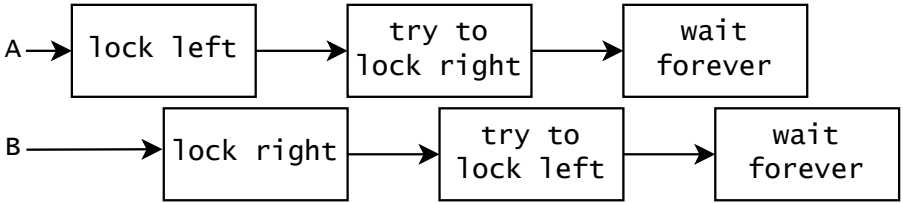
Fünf Philosophen ...



Deadlocks

Ein Deadlock bezeichnet eine Verklemmung des Systems

Hält Thread A eine Ressource, die Thread B gerne hätte, und Thread B eine Ressource, die Thread A gerne hätte, so können beide nicht weiterarbeiten und befinden sich in einem nicht auflösbaren Wartezustand.



Deadlocks

Beispiel für einen Deadlock

```
public class LeftRightDeadlock {
  private final Object left = new Object();
  private final Object right = new Object();

  public void leftRight() {
    synchronized (left) {
      synchronized (right) {
        doSomething();
      }
    }
  }
  public void rightLeft() {
    synchronized (right) {
      synchronized (left) {
        doSomething();
      }
    }
  }
}
```

Beispiel für einen Deadlock

```
public class Bank {

  static transferMoney(Account fromAcc, Account toAcc, Double amount) {
    synchronized(fromAcc) {
      synchronized(toAcc) {
        transfer(fromAcc, toAcc, amount);
      }
    }
  ...
  }

  public static void main (String[] args) {
    new Thread(() -> Bank.transferMoney(account1, account2, 100.)).start();
    new Thread(() -> Bank.transferMoney(account2, account1, 50.)).start();
  }
}
```

Vermeidungsstrategien

Synchronisation kann zu schwacher Performance und Deadlocks führen!

- Ressourcen immer in der gleichen Reihenfolge anfordern und freigeben; ist allerdings in komplexen Systemen nicht immer möglich.
- Ressourcen temporär wieder zurückgeben; der Klügere gibt nach.
- Synchronisation reduzieren.
  - Zugriffsmethoden nur synchronisieren, wenn notwendig.
  - *Double-Check*
  - *Open calls*

Left-Right-Deadlock

```
public class Bank {

  static transferMoney(Account fromAcc, Account toAcc, Double amount) {
    int fromHash = System.identityHashcode(fromAcc);
    int toHash = System.identityHashcode(toAcc);

    if (fromHash < toHash) {
      synchronized(fromAcc) {
        synchronized(toAcc) {
          transfer(fromAcc, toAcc, amount);
        }
      }
    } if (fromHash > toHash) {
      synchronized(toAcc) {
        synchronized(fromAcc){
          transfer(fromAcc, toAcc, amount);
        }
  ...
}
```

Dynamische Reihenfolge zur Vermeidung von Deadlocks

Vermeidungsstrategien

```
public class Bank {
  static transferMoney(Account fromAcc, Account toAcc, Double amount) {
    boolean isTransfered = false;
    while (!isTransfered) {
      if (fromAcc.lock.tryLock()) {
        try {
          if (toAcc.lock.tryLock()) {
            try {
              transfer(fromAccount, toAccount, amount);
              isTransfered = true;
            } finally {
              toAcc.lock.unlock();
            }
          }
        } finally {
          fromAcc.lock.unlock();
        }
    ...
  }
```

Der Klügere gibt nach.

Vermeidungsstrategien

```
public class Singleton {
  private static Singleton instance;

  private Singleton() {}

  public synchronized Singleton getInstance() {
    if(instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Singleton-Implementierung mit *Lazy Initialization*, schwache Performance

Vermeidungsstrategien

```
public class Singleton {
  private volatile static Singleton instance;

  private Singleton() {}

  public Singleton getInstance() {
    if(instance == null) {
      synchronized (Singleton.class) {
        if(instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Singleton-Implementierung mit *Lazy Initialization* und *Double-check*

Cooperating-Objects-Deadlock

Alienmethode

Aus der Sicht einer Klasse C ist eine Alienmethode eine Methode, deren Verhalten nicht vollständig durch C spezifiziert ist. Dazu gehören sowohl Methoden in anderen Klassen als auch überschreibbare Methoden (weder privat noch final) in C selbst.

Cooperating-Objects-Deadlock

Alienmethode

Aus der Sicht einer Klasse C ist eine Alienmethode eine Methode, deren Verhalten nicht vollständig durch C spezifiziert ist. Dazu gehören sowohl Methoden in anderen Klassen als auch überschreibbare Methoden (weder privat noch final) in C selbst.

Halten einer Sperre beim Aufruf einer Alienmethode

Der Aufruf einer Alienmethode mit einer gehaltenen Sperre birgt das Risiko eines Deadlocks oder eines unerwartet lange Haltens und Blockierens anderer Threads, die die gehaltene Sperre benötigen.

Open Call

Der Aufruf einer Methode ohne gehaltene Sperre wird *open call* genannt. Nutze open calls, um das Risiko für Deadlocks zu minimieren.

4.4 Kooperation von Threads

Wait und Notify

Synchronisation mit `void wait()` und `void notify()`:

- Zusätzlich zu der Sperre eines Objekts besitzt dieses eine Warteliste.
- Mithilfe der Warteliste können Threads auf bestimmte Ereignisse warten, bis sie weiterarbeiten (z.B. **Producer und Consumer**).
- Dabei dürfen `wait()` und `notify()` nur aufgerufen werden, wenn das Objekt bereits gesperrt ist.
- Ein Aufruf von `wait()` nimmt die Sperre temporär zurück und setzt den aufrufenden Thread in die Warteliste.
- Beim Aufruf von `notify()` wird ein beliebiger Thread der Warteliste aktiviert, um seine Arbeit fortzusetzen.
- Beim Aufruf von `notifyAll()` werden alle Threads der Warteliste aktiviert, um ihre Arbeit fortzusetzen.

Producer Consumer Pattern

Producer

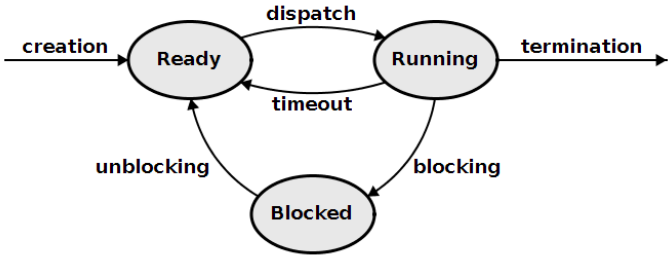
```
for (int i = 0; i < numProducer; i++) {
  new Thread(() -> {
    while (true) {
      synchronized (list) {
        while (list.size() == maxListSize) {
          try {
            list.wait();
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }

        list.add(new Data());
        list.notifyAll();
      }
    }
  }).start();
```

Producer Consumer Pattern

Consumer

```
for (int i = 0; i < numConsumer; i++) {
  new Thread(() -> {
    while (true) {
      synchronized (list) {
        while (list.size() == 0) {
          try {
            list.wait();
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
        }

        list.remove(0);
        list.notifyAll();
      }
    }
  }).start();
```

Zustandsdiagramm eines Threads



Ein Thread ist entweder **Bereit**, **Blockiert**, oder **in Ausführung**.

Beenden von Threads

Beenden von Threads:

- Die Methode `stop()` ist deprecated → Inkonsistente Zustände.
- Threads sollten stattdessen **kooperativ beendet** werden.
- Verwendung des Interrupt-Flags.

Kooperation kann nicht erzwungen werden.

4.5 Asynchrone Programmierung

Einführung

Motivation:

- Threads werden oft für die parallele Berechnung oder Anforderung von Daten verwendet.
- Diese Werte laufen in vielen Fällen wieder an einer **zentralen** Stelle zusammen.
- Ein **synchroner** Funktionsaufruf blockiert, bis die Funktion ausgeführt wurde.
- Die Synchronisation ist leider aufwendig und fehleranfällig.

Einführung

Motivation:

- Threads werden oft für die parallele Berechnung oder Anforderung von Daten verwendet.
- Diese Werte laufen in vielen Fällen wieder an einer **zentralen** Stelle zusammen.
- Ein **synchroner** Funktionsaufruf blockiert, bis die Funktion ausgeführt wurde.
- Die Synchronisation ist leider aufwendig und fehleranfällig.

Asynchrone Programmierung

- Ein **asynchroner** Funktionsaufruf blockiert nicht und gibt ein Stellvertreter-Objekt zurück.

Callables

Asynchrone Programmierung mit Callables

- Eine Klasse, die ein `Callable`-Objekt beschreibt, muss das Interface `interface Callable<T>` und somit die Methode `T call()` implementieren.
- Callables werden über einen `ExecutorService` gestartet.
- Der Rückgabewert wird zunächst in einem `Future`-Objekt gespeichert.
- Das `Future`-Objekt verfügt über die Methode `T get()`, mit welcher das Ergebnis des Threads geholt werden kann.
- Ist die Berechnung noch nicht fertig wird beim Aufruf von `get()` solange gewartet bis das Ergebnis vorliegt.

Die Methode `get()` sollte erst aufgerufen werden, nachdem alle Berechnungen gestartet sind.
Ansonsten keine Nebenläufigkeit!

Asynchrone Programmierung mit Callables

```
// Erstellen eines Executor Service
var exec = Executors.newCachedThreadPool();
List<Future<Integer>> futures = new ArrayList<>();
for (int i = 0; i < 10; ++i) {
  // Erstellen eines Callable Objects
  Callable<Integer> cCalc = () -> {
    return doVeryComplexCalculation();
  };
  // Submit and Execute!
  var future = exec.submit(cCalc) ;
  futures.add(future);
}
// Alle Ergebnisse anfordern!
int sum = 0;
for (var future: futures) {
  sum += future.get();
}
```

Unterschied Callables und Runnables

```
@FunctionalInterface
public interface Runnable {

  public abstract void run();
}
```

```
@FunctionalInterface
public interface Callable<V> {

  V call() throws Exception;
}
```

Unterschied Callables und Runnables

```
@FunctionalInterface
public interface Runnable {

  public abstract void run();
}
```

```
@FunctionalInterface
public interface Callable<V> {

  V call() throws Exception;
}
```

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

5. Interaktion mit Datenbanken

Universität Duisburg-Essen
Ingenieurwissenschaften
Informatik und Angewandte Kognitionswissenschaft
Intelligente Systeme
Josef Pauli

5.1 Einführung

Einführung

Motivation:

- Bei größeren Datenmengen bietet sich eine persistente Speicherung mit der **Serialisierung** nicht mehr unbedingt an.
- Daten **müssen** seriell eingelesen werden.
- Kein wahlfreier Zugriff.
- Daher werden in solchen Fällen oftmals **Datenbanken** verwendet.

Jeder **Datenbank** liegt ein **Datenbankmodell** zugrunde:

- Relational (tabellenbasiert, sehr verbreitet).
- Objektorientiert (Speicherung von Objekten).

Ein Datenbanksystem besteht aus mehreren Komponenten

Datenbankverwaltungssystem:

- Verantwortlich für die strukturierte Speicherung der Daten.
- Stellt kontrollierte lesende und schreibende Zugriffsmöglichkeiten bereit.
- Muss nicht zwangsläufig nach dem Client-Server Muster aufgebaut sein.

Datenbanksprache:

- Externe Schnittstelle zur Formulierung von Anfragen.

Datenbank:

- Beinhaltet die konkreten Daten.

Einführung Relationale Datenbanken

Relationale Datenbanken

Relationale Datenbanken:

- Tabelle mit Zeilen und Spalten
- Spalte: Relationsattribut
- Zeile: Relationstupel
- Inhalt der Datenbank: Menge der Relationstupel
- Jede Zeile benötigt einen **Primärschlüssel**

id	name	wohnort	strasse
1	Hoven G.	Linz	Ostweg 50
2	Baumgarten R.	Hannover	Am Tank 23
3	Strauch GmbH	Linz	Beerenweg 34a

Ein **(Primär)-Schlüssel** ist eine Gruppe von Spalten, die so ausgewählt wird, dass es keine zwei Zeilen mit der gleichen Kombination von Werten in dieser Spaltengruppe gibt.

Structured Query Language (SQL)

Structured Query Language (SQL):

- Definitions-, Manipulations- und Anfragesprache für relationale Datenbanken.
- Von IBM entwickelt seit Anfang der siebziger Jahre.
- Erste SQL-Norm vom ANSI-Konsortium in 1986 verabschiedet.
- Basiert auf der relationalen Algebra.
- SQL-Anweisungen sind nahezu umgangssprachlich formuliert.
- SQL-Anweisungen unabhängig von Groß-und Kleinschreibung, für bessere Lesbarkeit der SQL-Anfrage sind Kommandos großgeschrieben.
- Leerzeichen, Return und Tabulatoren sind bedeutungslos und nur für die bessere Lesbarkeit.
- SQL-Anweisungen werden in manchen Datenbanksystemen mit einem Semikolon abgeschlossen.

Structured Query Language (SQL)

DDL (Data Definition Language):

- Erstellen der Tabellen, Beziehungen und Indizes
- SQL-Anweisungen: `CREATE`/`DROP TABLE`, ...

DML (Data Manipulation Language):

- Daten hinzufügen und löschen
- SQL-Anweisungen: `INSERT INTO`, `DELETE`, `VALUES`, ...

DQL (Data Query Language):

- Daten auswählen und filtern
- SQL-Anweisungen: `SELECT`, `FROM`, `WHERE`, `ORDERED BY`, `EXISTS`, `JOIN`, ...

Data Definition Language:

```
CREATE TABLE lieferanten (
  id INTEGER PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(15),
  wohnort VARCHAR(15),
  strasse VARCHAR(15));
```

- `CREATE TABLE`: Neues Schema von Relation/Tabelle anlegen
- `PRIMARY KEY`: Primärschlüssel der Tabelle
- `INTEGER AUTO_INCREMENT` : Automatisch inkrementierter Integerwert
- `VARCHAR(15)` : Character-String der max. Länge 15

Data Manipulation Language :

```
INSERT INTO lieferanten (name,wohnort,strasse)
VALUES ('Hoven G.', 'Linz', 'Ostweg 50');
```

- `INSERT INTO`: Tabelle, in welche ein Relationstupel eingefügt wird
- `VALUES`: (Attribut-)Werte des einzufügenden Relationstupels

Data Query Language:

```
SELECT name
FROM lieferanten
WHERE wohnort='Linz';
```

- `SELECT`: Auswahl der Spalte(n) (Attribut)
- `FROM`: Auswahl der Tabelle(n) (Relation)
- `WHERE`: Auswahl der Zeile(n) (Relationstupel)

Ein `SELECT`-Statement kann mehrere Relationstupel zurückgeben!

5.2 Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC)

- Satz von Schnittstellen, um relationale Datenbanken in Java-Programmen zu nutzen.
- JDBC-API und zugehörige Implementierung (Datenbank-Treiber) liefern eine Abstraktion von relationalen Datenbanksystemen, d.h. mit einheitlicher Programmierschnittstelle können die differierenden Funktionen verschiedener Datenbankmanagementsysteme in gleicher Weise genutzt werden.
- Lernen von unterschiedlichen APIs für unterschiedliche Datenbanksysteme der Hersteller entfällt.
- Beispiele von Datenbanksystemen:
  - Kommerziell: Microsoft SQL Server, Oracle Database, ...
  - Open-Source: MySQL, PostgreSQL, ...

Java Database Connectivity (JDBC)

Schritte zur Datenbankabfrage:

- JDBC-Datenbanktreiber laden (deprecated)
- Verbindung zur Datenbank aufbauen
- SQL-Anweisung erzeugen
- SQL-Anweisung ausführen
- Ergebnis der Anweisung holen
- Datenbankverbindung schließen

Verbindung zur Datenbank aufbauen

- Nach dem Laden des Treibers wird mithilfe der Methode `getConnection()` der Klasse DriverManager eine Verbindung zur Datenbank aufgebaut.
- Diese Methode liefert ein Verbindungsobjekt der Klasse `java.sql.Connection` zurück, welches während der gesamten Verbindung besteht und als Lieferant für spezielle Objekte zur Abfrage und Veränderung der Datenbank dient.
- Der Methode wird der Pfad zu einer Datenbank mitgegeben.

```
try (var con = DriverManager.getConnection("jdbc:sqlite:database.db")) {
  ...
} catch (SQLException e) {
  e.printStackTrace();
}
```

SQL-Anweisung Erzeugen und Ausführen

Für die SQL-Anfragen und Manipulationen der Datenbank wird ein generisches Anweisungsobjekt `Statement` durch die Methode `createStatement` des Verbindungsobjektes erzeugt.

- Wichtigste Methoden sind `executeUpdate(String)` sowie `executeQuery(String)`
- `executeUpdate(String)` : zur Modifikation der Datenbank
  - **UPDATE**-, **INSERT**- oder **DELETE**-Anweisungen
- `executeQuery(String)` : für Anfragen an Datenbank mit Ergebnis-Rückgabe
  - **SELECT**-Anweisung
- Bei bestimmten Treibern sehr kostspielige Ressource.

Beispiel für die Verwendung von Statement

```
try (Statement stmt = con.createStatement()) {
  stmt.executeUpdate("INSERT INTO lieferanten (name, wohnort, strasse) "
    + " VALUES "
    + "('A','Linz','Ostweg 50'),"
    + "('B','Essen','Am Tank 23'),"
    + "('C','Linz','Beerenweg 34a'),"
    + "('D','Aalen','Hintergarten 9')");
} catch (SQLException e) {
  e.printStackTrace();
}
```

Ergebnis der Anweisung holen

- Bei Datenbankabfragen mit einem Rückgabewert wird ein `ResultSet` zurückgegeben.
- Ein `ResultSet` benötigt immer eine Datenbankverbindung.
- Es ist kein Container, indem die Daten lokal gehalten werden.

```
try (Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT id, name, wohnort FROM lieferanten")) {
  while (rs.next()) {
    int idVal = rs.getInt("id");
    String nameVal = rs.getString("name");
    String wohnortVal = rs.getString("wohnort");
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

Datenbankverbindung schließen

- Ein Objekt vom Typ `Connection` verfügt über die Methode `close()`, um die Verbindung zur Datenbank zu schließen.
- Das Schließen der Verbindung ist wichtig, daher sollte die Anweisung generell in einem **finally**-Block stehen oder ein **try-with-resources statement** verwendet werden.

```
try (Connection con = DriverManager.getConnection("jdbc:sqlite:database.db")) {
  // Do Stuff ...
} catch (SQLException e) {
  e.printStackTrace();
}
```

Transaktionen

Transaktionen

Motivation

Das oberste Ziel ist die Integrität und Konsistenz der Datenbank!

Definition

Bei einer **Transaktion** werden mehrere Anweisungen zusammengefasst.

- Eine **Transaktion** ist atomar, d.h. nicht unterbrechbar.
- Falls eine **Transaktion** scheitert, werden die Veränderungen rückgängig gemacht.
- Wird in JDBC für einzelne Anweisungen automatisch gemacht $\mapsto$ **Auto-Commit**.
- Falls eine Folge von Anweisungen als **eine** nicht unterbrechbare Transaktion gewünscht ist, so muss **Auto-Commit** zuerst abgeschaltet werden!

Sicherheit in Datenbanksystemen

Sicherheit in Datenbanksystemen

Datenbanksysteme müssen von Angriffen von **außen** gesichert werden!

- Daten müssen vor unbefugtem Zugriff geschützt werden.
- Sicherheit des Systems muss gewährleistet sein.

Sehr häufiges Problem: **SQL-Injection**-Angriff

- Einschleusung/Manipulation von SQL-Statements.
- Oft werden Eingabemöglichkeiten missbraucht.

Problem

SQL-Statements können mit Semikolons konkateniert werden.

Beispiel SQL-Injection-Lücke:

- Erwartete Eingabe: `42`
- Bösartige Eingabe: `42;UPDATE USER SET TYPE=admin WHERE ID=23;`

Lösung

Alle Benutzereingaben **müssen** kontrolliert werden!

Demonstration SQL-Injection-Lücke

```
String searchQuery = view.getInputPaneText();
try (Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT id, name, wohnort FROM lieferanten WHERE
      id = "+searchQuery)) {
  while (rs.next()) {
    int idVal = rs.getInt("id");
    String nameVal = rs.getString("name");
    String wohnortVal = rs.getString("wohnort");
    System.out.println("id = " + idVal + ", name = " + nameVal
      + ", wohnort = " + wohnortVal);
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

SQL-Injection-Lücken sind oft unerwartet!

Es soll ein automatisches System zur Erfassung von Fahrzeugen mit einem kamera-basierten Kennzeichen-Scanner realisiert werden. Was kann da schon schiefgehen?

SQL-Injection-Lücken sind oft unerwartet!

Es soll ein automatisches System zur Erfassung von Fahrzeugen mit einem kamera-basierten Kennzeichen-Scanner realisiert werden. Was kann da schon schiefgehen?

PreparedStatement:

- Prepared Statements sind parametrisierte SQL-Anweisungen.
- Sonderzeichen werden automatisch gefiltert.
- Bösartige Eingaben führen zu einem Fehler.
- Sie werden deklariert und vorkompiliert.
- Bei sich häufig wiederholenden Anfragen bieten Prepared Statements einen Geschwindigkeitsvorteil zur Laufzeit.

```
try (PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO lieferanten (name, wohnort) VALUES(?,?)")) {
  pstmt.setString(1, "Heinz");
  pstmt.setString(2, "Duisburg");
  pstmt.executeUpdate();
} catch (SQLException e) {
  e.printStackTrace();
}
```

5.3 Objektrelationale Abbildungen

Objektrelationale Abbildung

Beobachtung

- Objektorientierte Programmiersprachen kapseln Daten und Verhalten in Objekten.
- Relationale Datenbanken legen die Daten in Tabellen ab.
- Grundlegend verschiedene Paradigmen.
- **Object-Relational Impedance Mismatch.**

Lösungen

- Objektorientierte Datenbanken $\mapsto$ Keine Relationale Algebra.
- Objektrelationale Abbildung (ORM) $\mapsto$ Abbildungen nicht trivial.

Objektrelationale Abbildung

Umsetzung:

- Abbildung von Klassen auf Tabellen.
- Jedes Attribut wird durch eine Tabellenspalte repräsentiert.
- Jedes Objekt entspricht einer Tabellenzeile.
- Primärschlüssel $\mapsto$ Zeilentupel oder zusätzliche Spalte?
- Hat ein Objekt eine Referenz auf ein anderes Objekt, so kann diese mit einer Primärschlüssel-Fremdschlüssel-Beziehung in der Datenbank dargestellt werden.

Objektrelationale Abbildung

Umsetzung:

- Abbildung von Klassen auf Tabellen.
- Jedes Attribut wird durch eine Tabellenspalte repräsentiert.
- Jedes Objekt entspricht einer Tabellenzeile.
- Primärschlüssel $\mapsto$ Zeilentupel oder zusätzliche Spalte?
- Hat ein Objekt eine Referenz auf ein anderes Objekt, so kann diese mit einer Primärschlüssel-Fremdschlüssel-Beziehung in der Datenbank dargestellt werden.

Worin besteht jetzt die Schwierigkeit?

Abbildung von Vererbungshierarchien

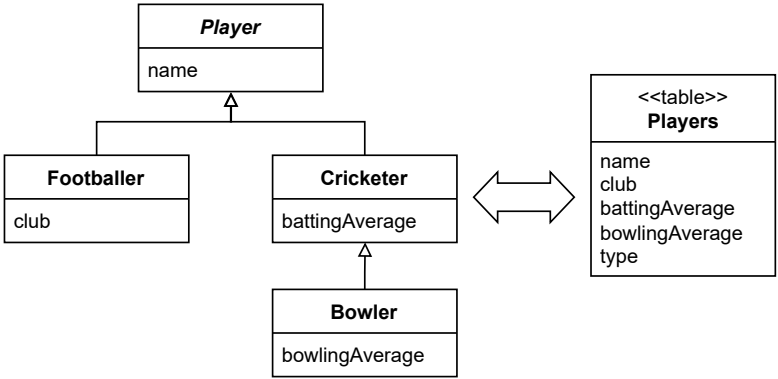


Abbildung: *Single Table Inheritance* - Eine Tabelle für gesamte Vererbungshierarchie

Abbildung von Vererbungshierarchien

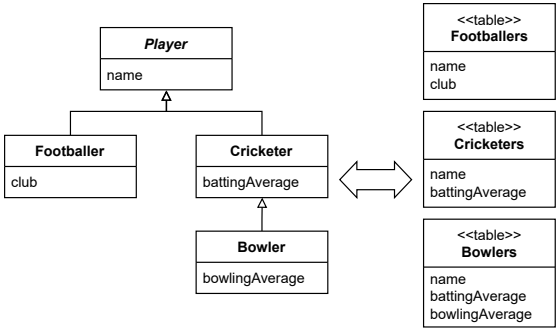


Abbildung: *Concrete Table Inheritance* - Je eine Tabelle für jede konkrete Klasse in Vererbungshierarchie

Abbildung von Vererbungshierarchien

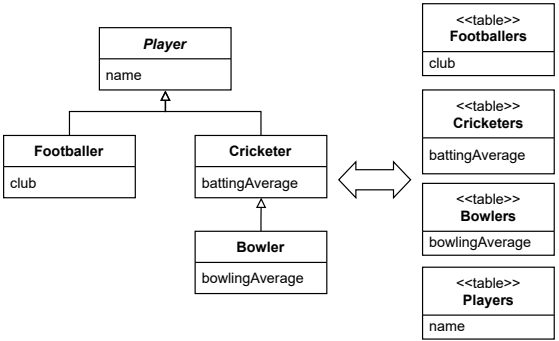


Abbildung: *Class Table Inheritance* - Je eine Tabelle für jede Klasse in Vererbungshierarchie

Objektrelationale Abbildung

Anforderungen

- Keine manuelle Interaktion mit der Datenbank.
- Kein manuelles Formulieren von SQL-Statements.
- Verwendung von **Annotationen**.
- Interne Implementierungsdetails werden wegabstrahiert.

Objektrelationale Abbildung

Anforderungen

- Keine manuelle Interaktion mit der Datenbank.
- Kein manuelles Formulieren von SQL-Statements.
- Verwendung von **Annotationen**.
- Interne Implementierungsdetails werden wegabstrahiert.

Java Persistence API (JPA)

- Schnittstellenstandard für die Zuordnung und die Übertragung von Objekten zu Datenbankeinträgen.
- Es gibt mehrere Implementierungen: EclipseLink, OpenJPA, ...

Objektrelationale Abbildung

Konfiguration

- Aufgrund der Automatisierung ist eine umfangreiche Konfiguration notwendig.
- Konfiguration meistens über die **persistence.xml** Datei.

Mögliche Parameter:

- **openjpa.ConnectionURL** $\mapsto$ `jdbc:mysql://localhost:3306/new_schema`
- **openjpa.ConnectionDriverName** $\mapsto$ `com.mysql.cj.jdbc.Driver:`
- **openjpa.jdbc.SynchronizeMappings** $\mapsto$ `add`, `refresh`, `buildSchema`

Objektrelationale Abbildung

Konfiguration

- Aufgrund der Automatisierung ist eine umfangreiche Konfiguration notwendig.
- Konfiguration meistens über die **persistence.xml** Datei.

Mögliche Parameter:

- **openjpa.ConnectionURL** $\mapsto$ `jdbc:mysql://localhost:3306/new_schema`
- **openjpa.ConnectionDriverName** $\mapsto$ `com.mysql.cj.jdbc.Driver:`
- **openjpa.jdbc.SynchronizeMappings** $\mapsto$ `add`, `refresh`, `buildSchema`

Die Konfiguration ist abhängig von Datenbank und JPA-Implementierung.

Beispiel für eine simple Klasse

```
@Entity
public class SimpleClass implements Serializable {
  private static final long serialVersionUID = 1;

  @Id
  @GeneratedValue
  private int id;

  public int getId();
  public void setId(int id);
}
```

- Verwendete Klassen müssen Serializable und mit @Entity gekennzeichnet sein.
- Der Primärschlüssel wird durch @Id markiert.
- Der Primärschlüssel kann durch @GeneratedValue automatisiert erstellt werden.

Beispiel für eine komplexere Klasse

```
@Entity
public class ComplexClass implements Serializable {
    private static final long serialVersionUID = 1;

    @Id
    @GeneratedValue
    private int id;

    @ElementCollection
    private List<SimpleClass> simpleObjectCollection = new ArrayList<>();

    @MapKey
    private Map<String, SimpleClass> simpleObjectMap = new HashMap<>();
}
```

Initialisierung

```
var managerFactory = OpenJPAPersistence.getEntityManagerFactory();
EntityManager entityManager = managerFactory.createEntityManager();
```

Persistierung von Objekten

```
EntityTransaction transaction = entityManager.getTransaction();
var complex = new ComplexClass();
transaction.begin();
entityManager.persist(complex);
transaction.commit();   // Commits the current transaction,
            // sets the id property of the persisted object
```

- ▷ Unterstützung von **Transaktionen** ↦ Alles oder nichts!
- ▷ Die **Id** wird automatisiert gesetzt und sollte nicht manuell verändert werden.

Wiederherstellen von Objekten

```
var query = entityManager.createQuery("SELECT x FROM ComplexClass x");
for (Object o : query.getResultList()) {
  ComplexClass complex = (ComplexClass) o;
}
```

```
entityManager.createQuery("SELECT x FROM ComplexClass x WHERE x.id = 134");
```

- Eine an SQL **angelehnte** Sprache steht zur Verfügung.
- Aufpassen beim Down-Casten!
- Die Anfragen können Bedingungen beinhalten.

Einschub Reflection

Definition von Reflection:

Reflection (oder **Introspektion**) bedeutet, dass ein Programm seine eigene Struktur kennt und diese eventuell modifizieren kann.

Es gibt mehrere Arten von Reflection:

- **Compile Time** Reflection.
- **Dynamische** Reflection: Zur **Laufzeit** stehen Informationen über Klassen und Objekte zur Verfügung.

Dynamische Reflection wird z.B. beim Casten von Objekten verwendet.
Falsches Casten von Objekten ⇒ ClassCastException!

Einschub Reflection

Class Klasse:

- Die Klasse `Class` gehört zur **Reflection API** und stellt Methoden zur Lieferung von Metadaten über Klassen und Interfaces zur Verfügung.
- Nur die JVM kann `Class`-Objekte erzeugen, da der Konstruktor private ist, außerdem sind die Objekte unveränderbar.

Primary Expression

```
Class<?> c1 = Data.class;
```

Objekt-Methode

```
Class<?> c2 = new Data().getClass();
```

Statische Factory Methode

```
Class<?> c3 = Class.forName("Data");
```

Einschub Reflection

Über `Class` -Objekte können Informationen über **Methoden**, **Attribute** und **Constructoren** angefordert werden:

- `Constructor<?>[] getConstructors()`
- `Method[] getMethods()`
- `Field[] getFields()`

```
var classObject = Class.forName("Data");
var constructor = classObject.getConstructor();
var data = constructor.newInstance();
```

```
var classObject = Data.class;
var constructor = classObject.getConstructor(String.class);
var data = constructor.newInstance("Useful String");
```

6. Netzwerkprogrammierung

Universität Duisburg-Essen
Ingenieurwissenschaften
Informatik und Angewandte Kognitionswissenschaft
Intelligente Systeme
Josef Pauli

6.1 Einführung

Motivation

Wo wird Netzwerkprogrammierung eingesetzt?

- Kommunikation in einem verteilten System
- Kommunikation zwischen lokalen Anwendungen
  → Einfacher als Shared Memory oder Dateien

Konkrete Beispiele:

- Robot Vision System
- Webservices
- Multiplayer

Begriffe zur Netzwerkprogrammierung

- **IP-Nummer:** Eine eindeutige Adresse, die jeden Host-Computer im Internet kennzeichnet.
  - **IPv4-Adresse:** 32-Bit-Zahl
  - **IPv6-Adresse:** 128-Bit-Zahl $\mapsto$ Weiterentwicklung von IPv4
- **Port (-Nummer):** Zusätzliche Adresskomponente von 16 Bits, die einen bestimmten Dienst in einem Netzwerk identifiziert.
- **Protokoll:** Regeln zur Festlegung von Identifikation, Format, Inhalt, Bedeutung und Reihenfolge gesendeter Nachrichten zwischen verschiedenen Computern.
- **Paket:** Nachrichten werden im Internet unterteilt in identifizierbare (Daten-Teil-)Pakete, im Netz befinden sich gleichzeitig verschiedene Pakete unterschiedlicher Nachrichten.
- **Router:** Host-Computer, der Pakete im Netzwerk bzw. zwischen Sub-Netzen weiterreicht.

Sockets

Sockets

Sockets (Steckverbindung) bilden eine **standardisierte** Schnittstelle zwischen der Protokoll-Implementierung im Betriebssystem und der eigentlichen Applikations-Software.

- Sockets sind Kommunikationsendpunkte.
- Sockets sind durch IP- und Port-Nummer eindeutig identifizierbar.
- Sockets werden verwendet, um Daten auszutauschen.
- Die Kommunikation erfolgt bidirektional.
- Sockets werden vom Betriebssystem verwaltet.

Prinzipiell wird zwischen **Stream-** und **Datagram-Sockets** unterschieden.

Sockets

Stream-Sockets:

- Benutzen (meist) das verbindungsorientierte **TCP/IP-Protokoll**.
- Die Verbindung zwischen den Rechnern bleibt die ganze Zeit während der Übertragung bestehen.
- Größere Datenmengen werden bei der Übertragung in kleine Pakete unterteilt und in der richtigen Reihenfolge übertragen.
- Es handelt sich um eine zuverlässige Übertragung, d.h. es ist gewährleistet, dass jedes Paket übertragen wird.
- Sicherheit vor Geschwindigkeit.
- Anwendungen: z.B. Online-Shops, Online-Banking.

Sockets

Datagram-Sockets:

- Benutzen (meist) das verbindungslose **UDP/IP-Protokoll**.
- Es wird **keine feste** Verbindung zwischen den Kommunikationspartnern aufgebaut.
- Datagramme werden einzeln verschickt und gelangen auf unterschiedlichen Wegen und in unterschiedlicher Reihenfolge zum Ziel.
- Die Übertragung ist nicht zuverlässig, d.h. es gibt keine Garantie, dass alle Pakete übertragen werden.
- Geschwindigkeit vor Sicherheit.
- Anwendungen: z.B. Abfrage von Börsendaten, Radiostreams.

6.2 Stream-Sockets in Java

Stream-Sockets in Java

Server:

- Der Server verfügt über einen `ServerSocket`.
- Durch die Methode `accept()` wartet der Server auf einen eingehenden Verbindungswunsch bzw. auf eingehende Verbindungswünsche.
- Bei einem Verbindungswunsch wird ein neuer `Socket` erzeugt, der als Basis für die Kommunikation dient.
- Das Socket-Objekt kann benutzt werden, um Streams für die Kommunikation zu erzeugen.
- Nach der Abarbeitung sollten die Streams geschlossen werden.

Um **mehrfache** Verbindungen abzuarbeiten, kann für jede Verbindung ein neuer Thread erstellt werden!

Stream-Sockets in Java

```
public class DaytimeServer {
  public static void main(String[] args) {
    try (ServerSocket server = new ServerSocket(13)) {
      while (true) {

        try (Socket connection = server.accept();
        var os = connection.getOutputStream();
        var oos = new ObjectOutputStream(os)) {

          oos.writeObject(new Date());
          oos.flush();

        } catch (IOException ex) {
          ex.printStackTrace();
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Stream-Sockets in Java

Client:

- Der Client versucht eine Verbindung mit einem `Socket` zum Server aufzubauen.
- Bei einer erfolgreichen Verbindung erzeugt der Client die Streams für den Server.
- Anschließend werden die Daten an den Server übermittelt.
- Nach Bearbeitung durch den Server und Rücksenden des Ergebnisses an den Client, wird es vom Client entgegengenommen.
- Wenn keine weiteren Anfragen anliegen, werden die Streams und die Verbindung geschlossen.

Stream-Sockets in Java

```
public class DaytimeClient {
  public static void main(String[] args) {
    while (true) {

      try (Socket socket = new Socket(InetAddress.getLocalHost(), 13);
      var is = socket.getInputStream();
      var ois = new ObjectInputStream(is)) {

        Date date = (Date) ois.readObject();
        System.out.println(date);

      } catch (IOException e) {
        e.printStackTrace();
      } catch (ClassNotFoundException e) {
        e.printStackTrace();
      }
    }
  }
}
```

6.3 Datagram-Sockets in Java

Datagram-Sockets in Java

Server:

- Der Server verfügt über ein `DatagramSocket` mit Information über welchen Port sein Dienst angeboten wird.
- Durch die Methode `receive()` wartet der Server auf eingehende Pakete der Klasse `DatagramPacket`.
- Kommt ein Paket an, kann es vom Server bearbeitet werden. Das Paket enthält die Daten des Absenders, damit der Server antworten kann.
- Für die Antwort bereitet der Server ebenfalls ein Paket vor und schickt es mit Methode `send()` an die Adresse des Client.
- Es wird zu keinem Zeitpunkt eine feste Verbindung aufgebaut, sondern es werden lediglich Pakete verschickt.

Datagram-Sockets in Java

Client:

- Der Client erstellt einen eigenen `DatagramSocket`, über welchen er senden und empfangen kann.
- Er erstellt ein Paket der Klasse `DatagramPacket` mit seiner Anfrage und schickt dieses an den Server mit Methode `send()`.
- Es wird gewartet mit Methode `receive()`, bis die Antwort vom Server eingeht.
- Es können weitere Pakete an den Server geschickt werden oder der Client arbeitet lokal mit den Daten weiter, da keine Verbindung besteht, müssen keine weiteren Schritte unternommen werden.

Der Server- und der Client-Socket sind Instanzen der **gleichen** Klasse.

Datagram-Sockets in Java

Schwierigkeiten bei der Verwendung von UDP:

- Die Reihenfolge der Pakete ist zufällig.
  → Sequenznummern
- Wie sicherstellen, dass alle Pakete ankommen?
  → Sequenznummern
- Wer muss nachfragen, wenn ein Paket fehlt?
  → Server oder Client
- Was passiert, wenn Anfragen doppelt gesendet werden?
  → Zeitstempel
- Daten sollten auch unvollständig einen Wert haben.
  → Jedes Paket beinhaltet einen Datensatz.
  → Manuelles Erstellen der Pakete.

Datagram-Sockets in Java

Schwierigkeiten bei der Verwendung von UDP:

- Die Reihenfolge der Pakete ist zufällig.
  → Sequenznummern
- Wie sicherstellen, dass alle Pakete ankommen?
  → Sequenznummern
- Wer muss nachfragen, wenn ein Paket fehlt?
  → Server oder Client
- Was passiert, wenn Anfragen doppelt gesendet werden?
  → Zeitstempel
- Daten sollten auch unvollständig einen Wert haben.
  → Jedes Paket beinhaltet einen Datensatz.
  → Manuelles Erstellen der Pakete.

Die korrekte Verwendung von **UDP** ist schwierig!

Einschub MSB und LSB

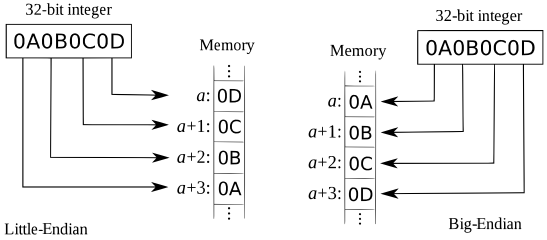
Primitive Datentypen bestehen aus mehreren Bytes/Bits

Dezimal:	23498
Hexadezimal:	0x5BCA (Zwei Ziffern = 1 Byte)
Binär (Zweierkomplement):	01011011 11001010

MSB = Most Significant Bit

LSB = Least Significant Bit

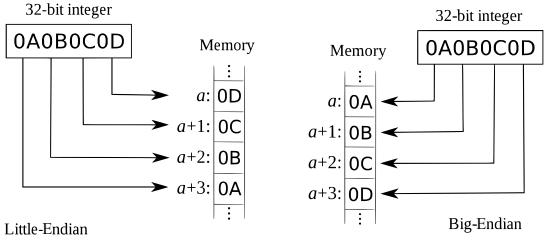
Byte-Reihenfolge (Endianness)



Big-Endian - Höchstwertige Byte zuerst

Little-Endian - Kleinstwertige Byte zuerst

Byte-Reihenfolge (Endianness)



Big-Endian - Höchstwertige Byte zuerst

Little-Endian - Kleinstwertige Byte zuerst

Beim **binären** Austausch von Daten muss die Endianness beachtet werden!

7. GUI Programmierung mit JavaFX

Universität Duisburg-Essen
Ingenieurwissenschaften
Informatik und Angewandte Kognitionswissenschaft
Intelligente Systeme
Josef Pauli

GUI Frameworks

- GUI ≡ Graphical User Interface ≡ Grafische Benutzerschnittstelle
- Es gab viele Versuche, ein GUI-Framework für Java zu erstellen ...

GUI Frameworks

Kurze Historie:

- ▷ JavaFX wurde erstmalig mit dem JDK 7 ausgeliefert
- ▷ War seit Java 8 fest integriert
- ▷ Ist mit Java 11 wieder **rausgeflogen** ...

Warum trotzdem JavaFX und nicht z.B. HTML5?

Bietet guten Überblick über alle relevanten Konzepte:

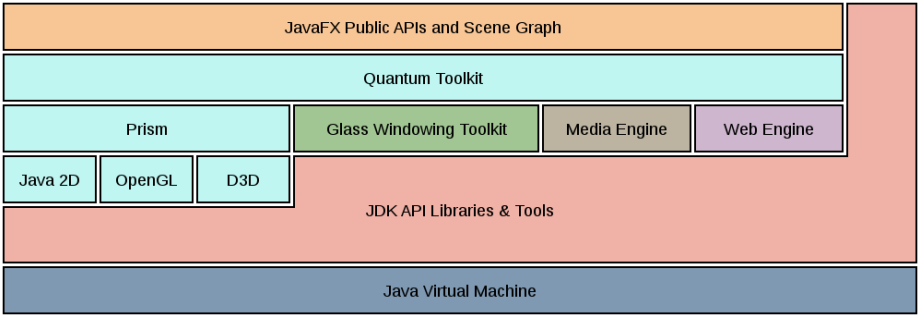
- \+ Unterstützt **hierarchischen** Aufbau der GUI
- \+ Verwendet das ereignisbasierte **Observer** Muster
- \+ Bietet Einblicke in **einige** interne Implementierungen

7.1 Einführung in JavaFX

Siehe auch:

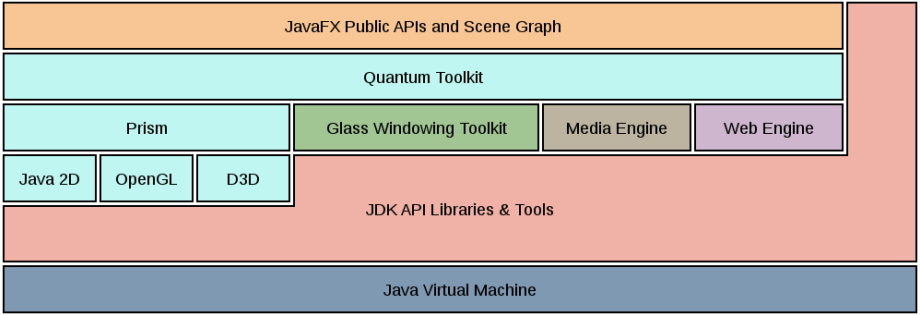
JavaFX Tutorial I
JavaFX Tutorial II
JavaFX Tutorial III

JavaFX Architektur



- Die **JVM** bildet die Grundlage jeder Java-Applikation
- **JDK** stellt Entwicklungstools bereit
- **Prism** ist die Rendering Engine für Grafiken
- **Glass Window Toolkit** für Low-Level-Betriebssystem-Routinen

JavaFX Architektur



- **Media Engine** zur Unterstützung von Audio und Video
- **Web Engine** ermöglicht Einbetten von Web-Inhalten
- **Quantum Toolkit** verküpft diese Elemente
- **JavaFX APIs** verbergen tiefer liegende Ebenen vor dem Programmierer

JavaFX GUI Aufbau

Eine JavaFX Anwendung ist hierarchisch aufgebaut und besteht aus **Stages, Scenes** und **Nodes** → Composite-Pattern

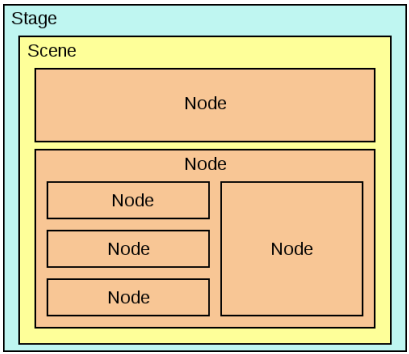
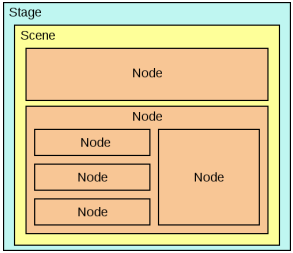


Abbildung: Schematischer Aufbau einer JavaFX-Oberfläche

Stage

javafx.stage.Stage

- Top-Level-JavaFX-Container
- Aussehen abhängig vom Betriebssystem
- In der GUI-Applikation wird zunächst via launch-Methode eine stage erstellt
- start-Methode erhält stage als Parameter und show-Methode stellt die stage dar

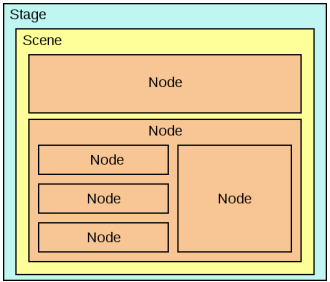


Stage ist die Bühne, auf der alles präsentiert wird

Scene

javafx.scene.Scene

- Container für Inhalte des Szenengraphen
- Verbindung zwischen Fenster (Betriebssystem) und Szenengraph (Java-Applikation)
- Szenengraph ist Baum der aus Knoten (Nodes) besteht
- Verwaltet Bestandteile der GUI

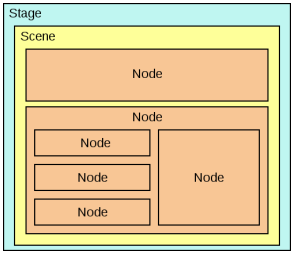


Scene die auf einer Bühne (Stage) aufgeführt wird

Node

javafx.scene.Node

- Bilden die Inhalte des Szenegraphs
- **Parent-Nodes** enthalten weitere Nodes als Kinder (evtl. selbst Parent von Kindern)
  - Unsichtbare Strukturelemente (z.B. BorderPane, HBox)
- **Leaf-Nodes** haben keine Kinder
  - Sichtbare GUI Elemente (z.B. Button, Label)

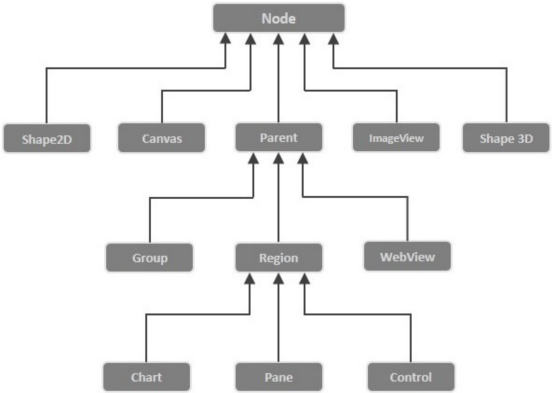


Nodes sind die Schauspieler in einer Scene

Node

Es gibt verschiedene Arten von Nodes, z.B.:

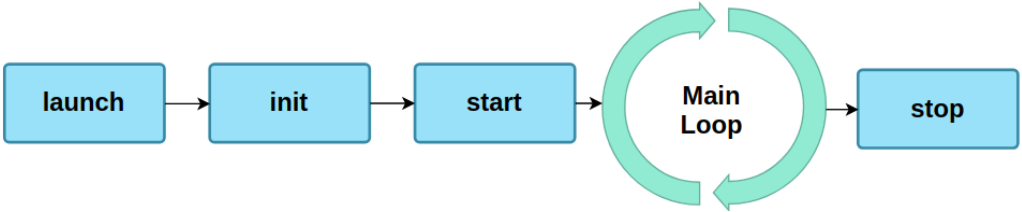
- **Control:** Basisklasse für UI Controls (z.B. Button, TextField, ChoiceBox, CheckBox, etc.)
- **Pane:** Basisklasse für Layouts (z.B. HBox, etc.)
- **Group:** Ermöglicht gleichzeitige Änderung an mehreren Elementen



Application

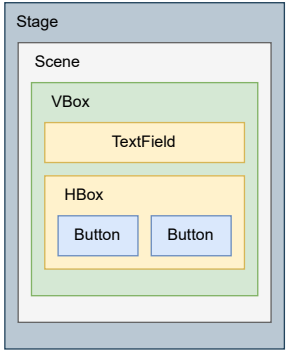
Application ist die Basisklasse für JavaFX Anwendungen und verwaltet den gesamten **Lebenszyklus** von JavaFX Anwendungen

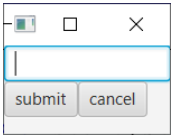
1) **launch:** Instanziierung der Application-Klasse
2) **init:** Verarbeitung von z.B. Eingabeparametern
3) **start:** Bekommt die PrimaryStage übergeben und beinhaltet somit den Aufbau der GUI
4) **Main-Loop:** Behandlung der Events
5) **stop:** Wird bei Beendigung der Applikation aufgerufen



Application

```
public class MinimalApplication extends Application {
  @Override
  public void start(Stage primaryStage) throws Exception {

    VBox layout = new VBox();

    TextField textField = new TextField();

    Button submitButton = new Button("submit");
    Button cancelButton = new Button("cancel");
    HBox buttons = new HBox(submitButton, cancelButton);

    layout.getChildren().addAll(textField, buttons);

    Scene scene = new Scene(layout);
    primaryStage.setScene(scene);
    primaryStage.show();
  }
}
```





7.2 Ereignisbehandlung in JavaFX

Ereignisbehandlung in JavaFX

JavaFX wendet das Observer Muster an:

- Objekte können Ereignisobjekte abfeuern (javafx.event.Event)
- Andere Objekte registrieren sich über Event-Listener
- Das Paket javafx.event enthält die relevanten Klassen

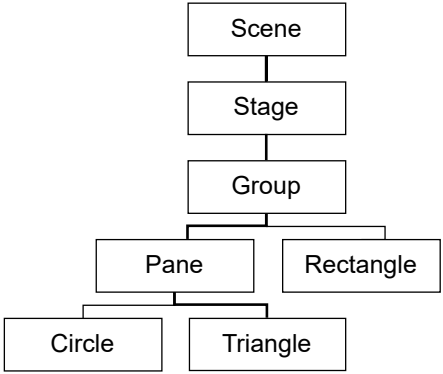
Ereignisbehandlung in JavaFX

Event-Capturing-Phase

In der *event capturing phase* wird das Ereignis vom Wurzelknoten ihrer Anwendung versendet und über die Ereigniskette an den Zielknoten weitergegeben.

Event-Bubbling-Phase

Nachdem das Ereignisziel erreicht ist und alle registrierten Filter das Ereignis verarbeitet haben, kehrt das Ereignis entlang der Ereigniskette vom Ziel zum Wurzelknoten zurück.



Ereignisbehandlung in JavaFX

Die Ereignisbehandlung erfolgt durch Filter und Handler, die Implementierungen der EventHandler-Schnittstelle sind.

Ereignisbehandlung in JavaFX

Die Ereignisbehandlung erfolgt durch Filter und Handler, die Implementierungen der EventHandler-Schnittstelle sind.

Event-Filter

Ein Event-Filter wird während der Event-Capturing-Phase ausgeführt. Ein Event-Filter für einen übergeordneten Knoten kann eine gemeinsame Ereignisverarbeitung für mehrere untergeordnete Knoten bereitstellen und, falls gewünscht, das Ereignis konsumieren, um zu verhindern, dass der untergeordnete Knoten das Ereignis erhält.

Ereignisbehandlung in JavaFX

Die Ereignisbehandlung erfolgt durch Filter und Handler, die Implementierungen der EventHandler-Schnittstelle sind.

Event-Filter

Ein Event-Filter wird während der Event-Capturing-Phase ausgeführt. Ein Event-Filter für einen übergeordneten Knoten kann eine gemeinsame Ereignisverarbeitung für mehrere untergeordnete Knoten bereitstellen und, falls gewünscht, das Ereignis konsumieren, um zu verhindern, dass der untergeordnete Knoten das Ereignis erhält.

Event-Handler

Ein Event-Handler wird während der Event-Bubbling-Phase ausgeführt. Wenn ein Event-Handler für einen untergeordneten Knoten das Ereignis nicht verbraucht, kann ein Event-Handler für einen übergeordneten Knoten auf das Ereignis reagieren, nachdem ein untergeordneter Knoten es verarbeitet hat, und kann eine gemeinsame Ereignisverarbeitung für mehrere untergeordnete Knoten bereitstellen.

Events konsumieren

Ein Ereignis kann von einem Event-Filter oder einem Event-Handler an einem beliebigen Punkt in der Ereigniskette konsumiert werden, indem die Methode `consume()` aufgerufen wird. Diese Methode signalisiert, dass die Verarbeitung des Ereignisses abgeschlossen ist und der Durchlauf der Ereigniskette endet.

Ereignisbehandlung in JavaFX

Events konsumieren

Ein Ereignis kann von einem Event-Filter oder einem Event-Handler an einem beliebigen Punkt in der Ereigniskette konsumiert werden, indem die Methode `consume()` aufgerufen wird. Diese Methode signalisiert, dass die Verarbeitung des Ereignisses abgeschlossen ist und der Durchlauf der Ereigniskette endet.

Hinweis

Standard-Handler für die UI-Controls von JavaFX konsumieren normalerweise die meisten Eingabeereignisse.

Ereignisbehandlung in JavaFX

Die Handler/Listener können mit Klassen, Anonymen Klassen oder Lambda Ausdrücken umgesetzt werden

```
Button button = new Button("Press Me!");
button.setOnAction(new EventHandler<ActionEvent>() {
  @Override
  public void handle(ActionEvent event) {
      System.out.println("Pressed!");
    }
});
```

```
button.setOnMouseEntered(event ->  System.out.println("Mouse Entered!"));
```

Ereignisbehandlung in JavaFX

Die Handler/Listener können mit Klassen, Anonymen Klassen oder Lambda Ausdrücken umgesetzt werden

```
Button button = new Button("Press Me!");
button.setOnAction(new EventHandler<ActionEvent>() {
  @Override
  public void handle(ActionEvent event) {
      System.out.println("Pressed!");
    }
});
```

```
button.setOnMouseEntered(event ->  System.out.println("Mouse Entered!"));
```

Wird ein Handler mit einer **setOn[...]** Methode gesetzt, so wird der vorherige Handler überschrieben!

Ereignisbehandlung in JavaFX

Mit der Methode **addEventHandler** können unterschiedliche Handler für unterschiedliche Ereignistypen registriert werden

```
button.addEventHandler(MouseEvent.MOUSE_ENTERED, new EventHandler<MouseEvent>() {
  @Override
  public void handle(MouseEvent event) {
    System.out.println("Mouse Entered!");
  }
});
```

```
button.addEventHandler(ActionEvent.ACTION, event -> {
  System.out.println("Button Pressed!");
});
```

Ereignisbehandlung in JavaFX

Observables:

- Auf **Properties** können sich **Listeners** registrieren
- Sobald sich der Wert eines **Properties** ändert, werden alle **Listener** benachrichtigt

```
StringProperty stringProperty = new SimpleStringProperty();
stringProperty.addListener(
  (observable, oldValue, newValue) -> {
      System.out.println("Value changed!");
});
```

Ereignisbehandlung in JavaFX

Observables:

- Auf **Properties** können sich **Listeners** registrieren
- Sobald sich der Wert eines **Properties** ändert, werden alle **Listener** benachrichtigt

```
StringProperty stringProperty = new SimpleStringProperty();
stringProperty.addListener(
  (observable, oldValue, newValue) -> {
      System.out.println("Value changed!");
});
```

Achtung!

- Alle Handler und Listeners werden im GUI-Thread ausgeführt
- Keine teuren Berechnungen anstoßen
- Auslagerung in andere Threads

7.3 Model-View-Controller
(Architekturmuster)

Beobachtung

- Benutzerschnittstellen sind **häufig** Änderungen unterworfen
- Oftmals gibt es nur einen **Prototypen**
- **Verschiedene** Benutzeroberflächen sollten integrierbar sein

Die Entwicklung und Wartung ist sehr **aufwändig** und **teuer**, wenn die Benutzeroberfläche stark mit dem funktionalen Kern zusammenhängt ...

Kapselung der verschiedenen Aufgaben

Model:

- Repräsentiert die **Daten**
- Stellt die **Kernfunktionalitäten** bereit
- Unabhängig von Ausgabeformaten oder Eingabemöglichkeiten

View:

- **Darstellung** der Daten aus dem Model
- View wird durch Controller festgelegt

Controller:

- Nimmt Befehle des Nutzers über die View entgegen
- Führt **Manipulationen** am Model oder View aus

Model-View-Controller

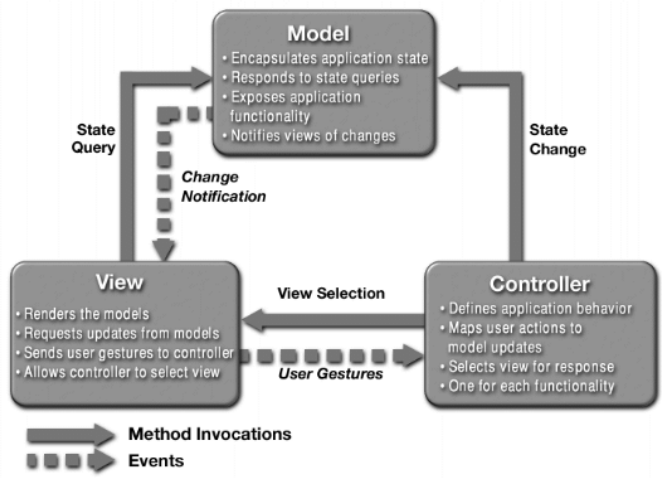


Abbildung: Schematische Darstellung von MVC (Quelle)

Einführung MVC

Anforderungen an Applikationen mit GUI-Komponente:

- Informationen sollen unterschiedlich dargestellt werden können
- Datenänderungen sollen sich unmittelbar in der Darstellung und im Verhalten der Anwendung widerspiegeln
- Eine Änderung der Benutzeroberfläche soll leicht durchzuführen sein, eventuell sogar zur Systemlaufzeit
- Der Code im Kern sollte unabhängig vom Erscheinungsbild und Benutzungsmerkmalen, sowie der Portierung der Benutzeroberfläche sein

Model-View-Controller

Historie:

- **MVC** in der Urform von **1979** war ein Meilenstein
- Ist noch immer das Standardmodell
- Viele Sprachen und Frameworks haben leicht angepasste Versionen

Es gibt viele Alternativen und Weiterentwicklungen:

- Hierarchical Model-View-Controller
- Model-View-Presenter
- Model-View-ViewModel
- Model-View-Presenter-ViewModel
- MVC und Alternativen I
- MVC und Alternativen II