



# Java Aufbau

Fortgeschrittene Programmierung



Cegos Group

inspire  
qualify  
change



# Inhalts- verzeichnis



Reflection



Einführung in das Java Persistence  
API



Generics



Context & Dependency Injection



Annotations



Funktionale Programmierung






Java und XML



# Reflection



-  Klassenobjekt und Reflection API
-  Dynamic Proxies
-  Dynamische Programmierung



# Klassenobjekt und Reflection API

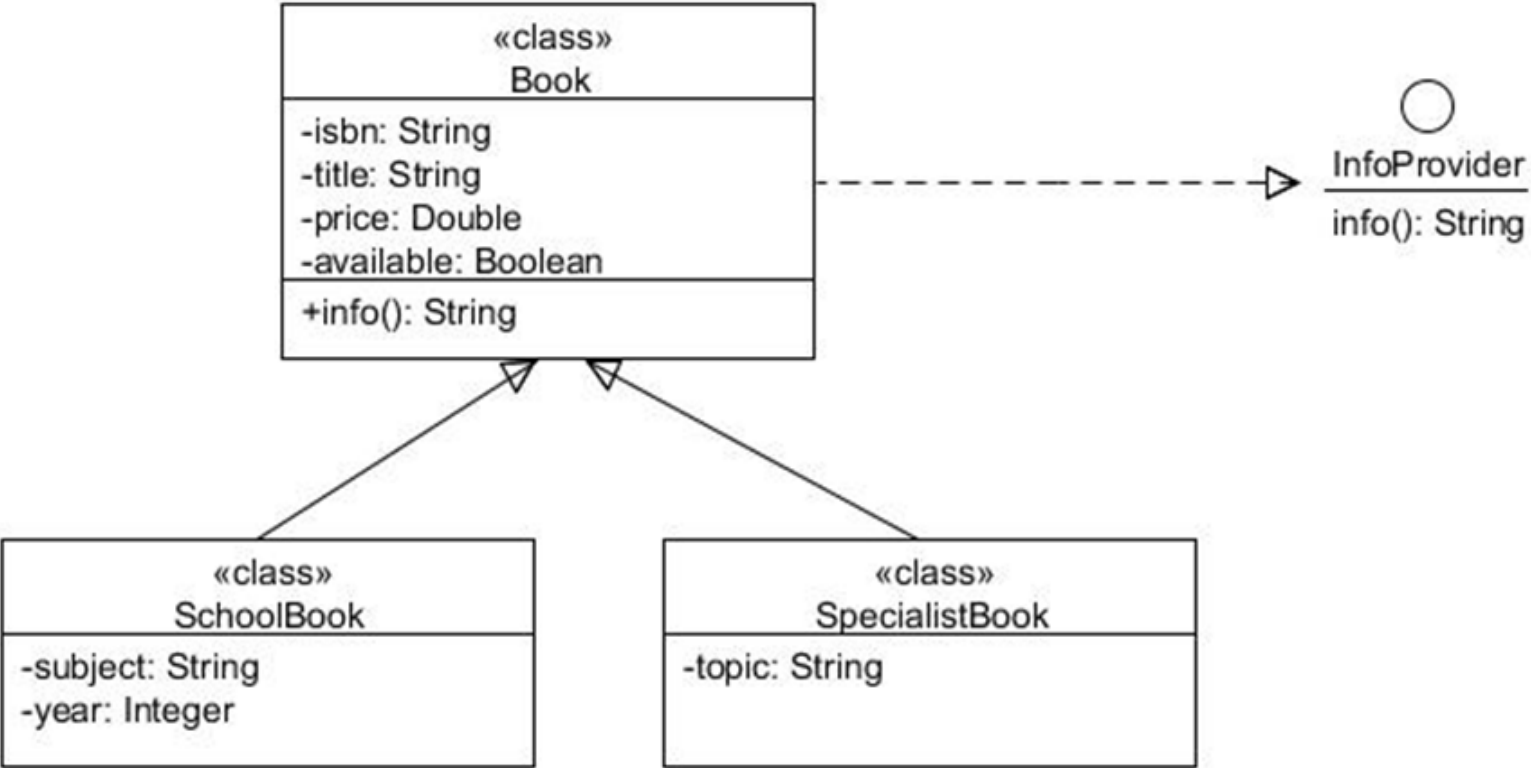


# Das Klassenobjekt

- In Java hat jedes Objekt einen fixen Satz von Laufzeittypen
  - Der Basis-Typ wird durch den Konstruktor eindeutig definiert
  - Alle weiteren Typen ergeben sich durch die Vererbungs- und Implementierungs-Hierarchie der Klassenmodellierung
- Die Java Virtual Machine löst zur Laufzeit jede Klassendefinition in ein Klassenobjekt auf
  - Dies passiert automatisch bei der ersten direkten oder indirekten Benutzung einer Klasse
  - Die `.class`-Datei mit dem Bytecode wird zum Klassenobjekt im Heap-Speicher der JVM

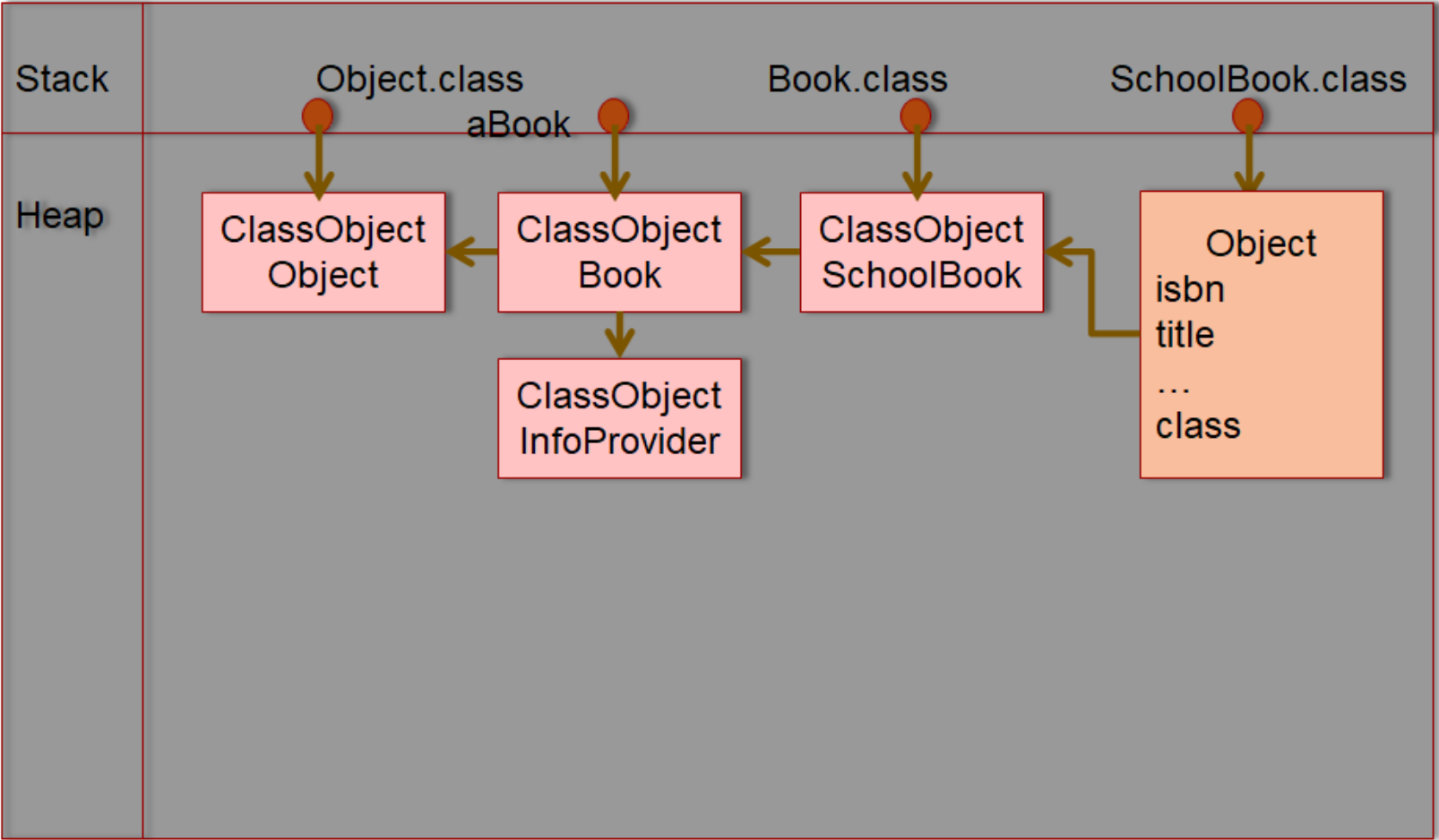


# Eine einfache Vererbungshierarchie





# Klassenobjekte im Speicher der JVM





# Bestandteile des Klassenobjekts

- Statische Elemente
  - Eigenschaften
  - Methoden
  - Initializer
    - `static`-Blöcke
    - Nur sichtbar während des Ladens der Klasse
- Object-Methoden
  - Nur die Eigenschaften sind dem Objekt zugeordnet
  - Objekt-Methoden benutzen zum Referenzieren des Objekts die bekannte `this`-Referenz
- Konstrukturen
  - Nur sichtbar für den `new`-Operator





# Auflösen von Eigenschaften und Methoden

- Die JVM sucht Eigenschaften ausschließlich im Objekt
- Statische Elemente sind nur durch die Klassenreferenz oder direkt durch die `class`-Eigenschaft des Objektes benutzbar
  - Damit werden statische Elemente nicht vererbt
    - Insbesondere gibt es keine Polymorphie
- Methoden werden durch die verkettete Liste der Superklassen hinweg gesucht
  - Der erste Treffer wird ausgeführt
  - Damit ist Polymorphie umgesetzt



# Das Reflection-API

- Programmatischer Zugriff auf die Struktur des Klassenobjekts
- Die Benutzung des APIs ist nicht Objekt-orientiert
  - Sieht eher etwas nach C aus
  - Beispiel
    - `method.invoke(delegate, args);`
- Das Reflection-API wurde schon früh in Java aufgenommen, hat sich aber nicht wesentlich verändert
  - Mit Java 5 wurden Annotations als zusätzlich Elemente aufgenommen
  - Leider auf Checked Exceptions ausgerichtet
    - Das Exception Handling ist damit immer aufwändig



# Javadoc

java.beans.beancontext  
java.io  
java.lang  
java.lang.annotation  
java.lang.instrument  
java.lang.invoke  
java.lang.management  
java.lang.ref  
java.lang.reflect  
java.math  
java.net  
java.nio  
java.nio.channels  
java.nio.channels.spi  
java.nio.charset  
java.nio.charset.spi  
java.nio.file

java.lang.reflect

Interfaces

AnnotatedArrayType  
AnnotatedElement  
AnnotatedParameterizedType  
AnnotatedType  
AnnotatedTypeVariable  
AnnotatedWildcardType  
GenericArrayType  
GenericDeclaration  
InvocationHandler  
Member  
ParameterizedType  
Type  
TypeVariable  
WildcardType

Classes

AccessibleObject  
Array  
Constructor  
Executable  
Field  
Method  
Modifier  
Parameter  
Proxy  
ReflectPermission

Exceptions

InvocationTargetException  
MalformedParameterizedTypeException  
MalformedParametersException  
NoSuchMethodException

Errors

GenericSignatureFormatError

Interface Summary

Interface	Description
AnnotatedArrayType	AnnotatedArrayType represents the potentially annotated use of an array type, whose component type may itself represent the annotated use of a type.
AnnotatedElement	Represents an annotated element of the program currently running in this VM.
AnnotatedParameterizedType	AnnotatedParameterizedType represents the potentially annotated use of a parameterized type, whose type arguments may themselves represent annotated uses of types.
AnnotatedType	AnnotatedType represents the potentially annotated use of a type in the program currently running in this VM.
AnnotatedTypeVariable	AnnotatedTypeVariable represents the potentially annotated use of a type variable, whose declaration may have bounds which themselves represent annotated uses of types.
AnnotatedWildcardType	AnnotatedWildcardType represents the potentially annotated use of a wildcard type argument, whose upper or lower bounds may themselves represent annotated uses of types.
GenericArrayType	GenericArrayType represents an array type whose component type is either a parameterized type or a type variable.
GenericDeclaration	A common interface for all entities that declare type variables.
InvocationHandler	InvocationHandler is the interface implemented by the invocation handler of a proxy instance.
Member	Member is an interface that reflects identifying information about a single member (a field or a method) or a constructor.
ParameterizedType	ParameterizedType represents a parameterized type such as Collection<String>.
Type	Type is the common superinterface for all types in the Java programming language.
TypeVariable<D extends GenericDeclaration>	TypeVariable is the common superinterface for type variables of kinds.
WildcardType	WildcardType represents a wildcard type expression, such as ?, * extends Number, or ? super Integer.

Class Summary

Class	Description
AccessibleObject	The AccessibleObject class is the base class for Field, Method and Constructor objects.
Array	The Array class provides static methods to dynamically create and access Java arrays.
Constructor<T>	Constructor provides information about, and access to, a single constructor for a class.
Executable	A shared superclass for the common functionality of Method and Constructor.
Field	A Field provides information about, and dynamic access to, a single field of a class or an interface.
Method	A Method provides information about, and access to, a single method on a class or interface.
Modifier	The Modifier class provides static methods and constants to decode class and member access modifiers.
Parameter	Information about method parameters.
Proxy	Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.
ReflectPermission	The Permission class for reflective operations.

2.0.0820 © Javacream

Java Aufbau

11



## Typische Aufgaben von Reflection

- Laden von Klassen über ihren Klassennamen
  - Sehr interessant für dynamisches Klassenladen und damit zur Entkopplung von Anwendungen
- Introspektion von Klassen
  - Welche Eigenschaften und Methoden sind definiert?
    - In dynamischen Script-Sprachen hat sich hierfür der Begriff "Duck Typing" eingebürgert
- Erzeugen von Objekten
  - Eine generisch konzipierte Anwendung kann damit Objekte erzeugen, deren Klassen ursprünglich noch gar nicht bekannt waren
- Zugriff auf Attribute
  - Sogar private Attribute sind via Reflection les- und schreibbar
    - Dazu muss das Programm aber die `java.lang.reflect.ReflectPermission` besitzen
- Aufrufen von Methoden



## Beispiel: Eine simple Factory

- Die Factory erzeugt Collection-Implementierungen
- Der zu benutzende Typ wird über eine Properties-Datei gelesen



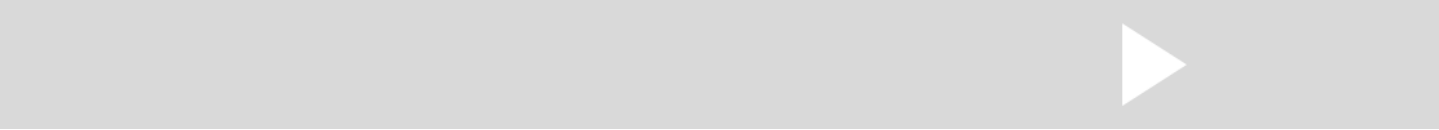
## Code: Eine simple Factory

```
public class SimpleFactory {
    private Properties properties;
    {
        properties = new Properties();
        properties.load(this.getClass().
            getResourceAsStream("simpleFactory.properties"));
    }
    public List<?> list() {
        Class<?> listClass =
            Class.forName(properties.getProperty("list"));
        List<?> instance = (List<?>) listClass.newInstance();
        return instance;
    }
    //...
```

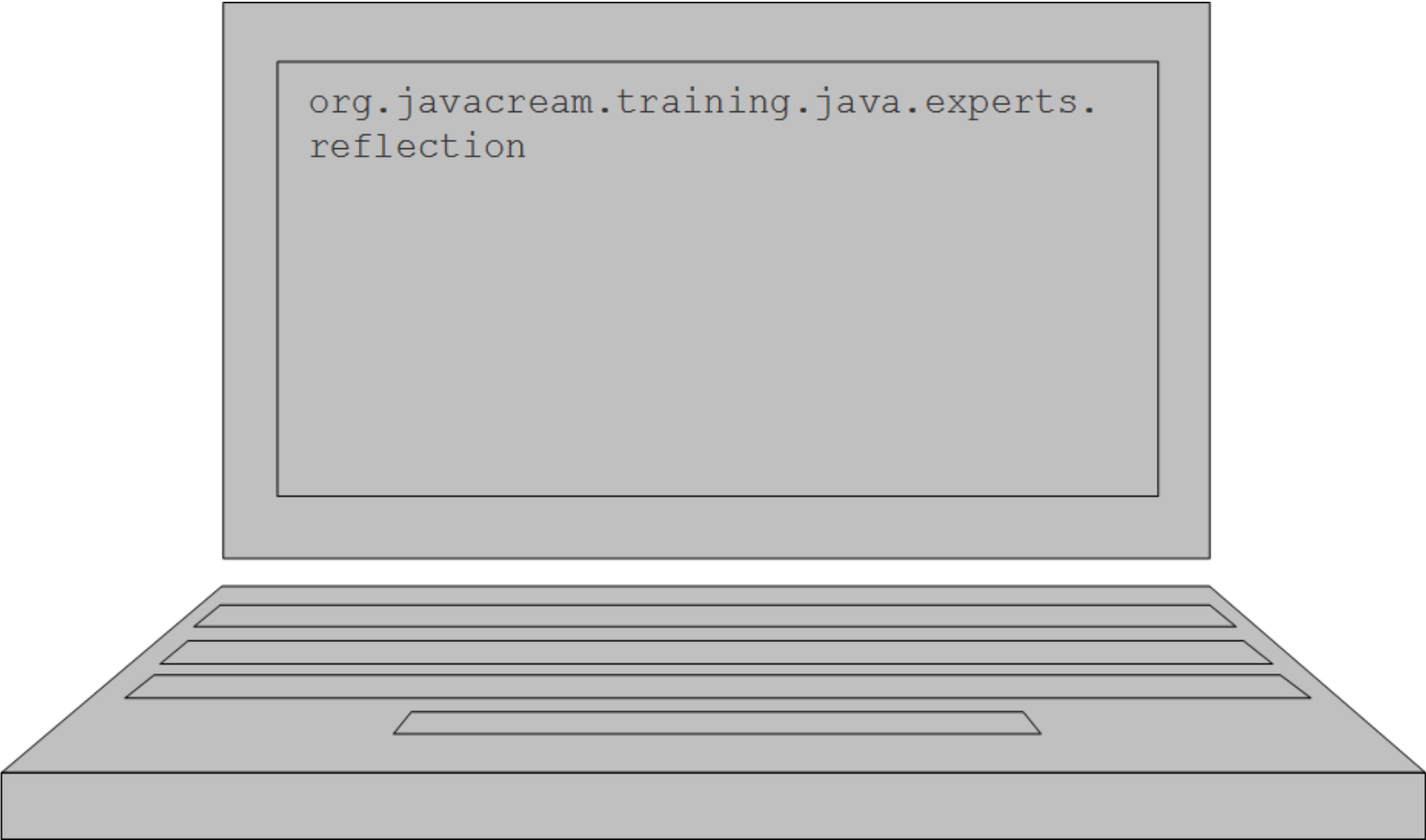


# Reflection und Overhead

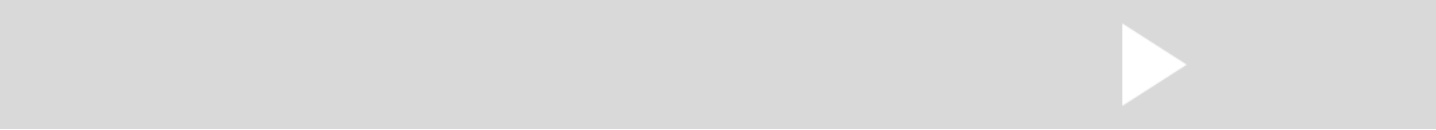
- Reflection-Aufrufe sind natürlich langsamer als direkt compilierte Methodenaufrufe
  - Beispiel:
    - `testPerformanceNoReflection: 0,002sec`
    - `testPerformanceWithReflection: 2,000sec`
- Die Auswirkungen sind in der Praxis aber eher gering
  - Beim obigen Test wurden jeweils 100.000.000 Aufrufe durchgeführt
  - Moderne Java Virtual Machines sind auch auf Reflection hin optimiert worden
- Die Auswirkungen auf den Speicherverbrauch sind ebenfalls moderat
  - Klassenobjekte sind bereits vorhanden
  - Reflection-Objekte wie Method etc. werden ab Java 8 im normalen Heap-Speicher abgelegt und sind damit für die Garbage Collection nichts besonderes mehr
    - Vorher: "Permanent Generation" mit dem gefürchteten Fehler `"OutOfMemoryError: Perm Space"`



# Hands On!







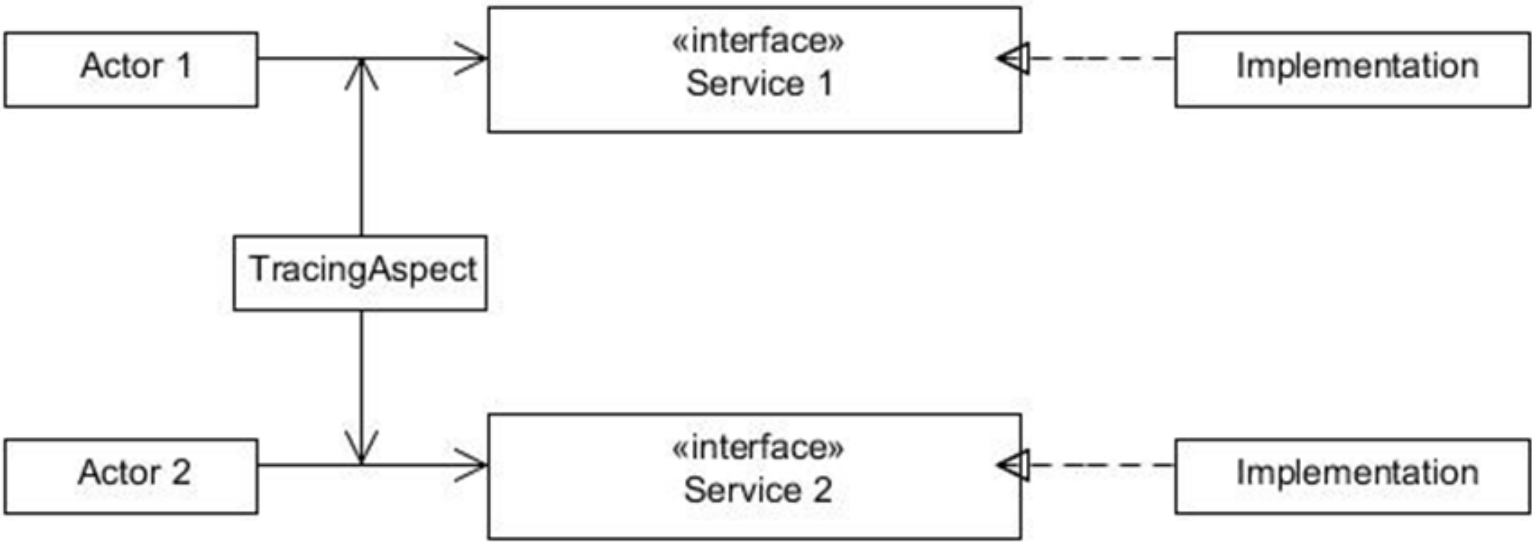
# Dynamic Proxies



# Problemstellung

- Im Rahmen der Software-Entwicklung werden häufig sogenannte "Querschnittsfunktionen" identifiziert
  - Englisch: Cross Cutting Concerns
  - Grundelement der Aspektorientierten Programmierung
- Diese sind für verschiedene Use Cases gleichermaßen gültig
  - und sollten deshalb zentral implementiert werden, um Code-Redundanzen zu vermeiden
- Beispiele:
  - Tracing
    - `entering, returning, throwing`
  - Authentifizierung
  - Überwachung
  - ...

# Beispiel





## Umsetzung in Java

- Die Umsetzung in Java scheint einfach zu sein:
  - Vererbung!
- Im Detail eröffnen sich jedoch sehr schnell Probleme
  - Statische Vererbungshierarchie
  - Finale Klassen
- Besser: Benutzung von Interfaces
  - Damit können Aspekte in einer Liste beliebig angeordnet werden
- Das größte Problem ist aber, dass durch die statische Typisierung in Java jeder Aspekt für jeden Use Case nochmals implementiert werden muss
  - Enormer Aufwand und damit ein
  - Wartungs Albtraum



## Lösung: Dynamic Proxies

- Das Reflection API stellt zwei Typen zur Verfügung, um dieses Problem zu lösen:
  - Die Schnittstelle `InvocationHandler`
    - Delegiert an ein beliebiges Objekt weiter und ruft eine (prinzipiell) beliebige Methode auf
  - Die Utility-Klasse `Proxy`
    - Diese erzeugt zur Laufzeit eine Implementierung einer beliebigen Schnittstelle
    - Wichtig: Proxy kann nicht mit Klassen umgehen!
    - und delegiert via Reflection alle Aufrufe an den bei der Erzeugung angegebenen `InvocationHandler` weiter



# Code: Der TracingAspect

```
public class TracingAspect implements InvocationHandler {
    Object delegate;

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        String methodName = method.getName();
        System.out.println("Entering " + methodName);
        try { Object result = method.invoke(delegate, args);
            System.out.println("returning from " + methodName);
            return result;}
        catch (Throwable e) {if (e instanceof InvocationTargetException) {
            e = ((InvocationTargetException) e).getTargetException();}
            System.out.println("throwing from " + methodName);
            throw e;}
    }

    public static Object createAspects(Object toDecorate){
        ClassLoader classLoader = TracingAspect.class.getClassLoader();
        Class<?>[] interfacesToImplement = toDecorate.getClass().getInterfaces();
        TracingAspect tracingAspect = new TracingAspect();
        tracingAspect.delegate=toDecorate;
        return Proxy.newProxyInstance(classLoader, interfacesToImplement, tracingAspect);
    }
}
```



## Code: Benutzung des TracingAspect

```
public class DynamicProxyTests {  
    @Test public void testDynamicProxy() {  
        List<String> names = new ArrayList<>();  
        names =  
        (List)TracingAspect.createAspects(names);  
        names.add("Hugo");  
        names.add("Emil");  
        names.add("Egon");  
        names.size();  
    }  
}
```



Code: "Ich kann  
alles!"

```
public class Omnipotent implements InvocationHandler {  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        Class<?> returnType = method.getReturnType();  
        switch (returnType.getName()) {  
            case "int":  
                return 42;  
            case "java.lang.String":  
                return "Hugo";  
            case "boolean":  
                return true;  
            //...  
            default:  
                return returnType.newInstance();  
        }  
    }  
    @SuppressWarnings("unchecked")  
    public static Object create(Class... interfaceTypes) {  
        ClassLoader classLoader = Omnipotent.class.getClassLoader();  
        Omnipotent omnipotent = new Omnipotent();  
        return Proxy.newProxyInstance(classLoader, interfaceTypes, omnipotent);  
    }  
}
```





# Dynamische Programmierung



## Java ist statisch typisiert

- Damit haben Entwickler eine ganze Reihe von Vorteilen:
  - Ein Compiler kann elementare Fehler prüfen
  - Eine Entwicklungsumgebung kann mit dem Autovervollständigungs-Feature ein sehr flüssiges Arbeiten ermöglichen
- Es gibt aber auch Probleme:
  - Ein Objekt kann zur Laufzeit sein Verhalten nicht erweitern
    - Eine Änderung ist durch die Einführung von "Strategies" möglich
      - Ein etabliertes Design Pattern
    - Methoden können auch nicht entfernt werden
  - Der Typ des Objekts ist nach seiner Erzeugung unveränderbar



# Dynamische Programmiersprache n

- Genau anders herum ist es bei dynamischen untypisierten Sprachen wie beispielsweise JavaScript
  - Hier gibt es eine feste Typisierung nur optional
  - Jedem Objekt können zur Laufzeit beliebige Eigenschaften hinzugefügt oder genommen werden
    - Auch Funktionen/Methoden sind Eigenschaften



# Dynamische Programmierung in Java?

- Mit Hilfe von Reflection kann Java dynamisch verwendet werden
  - Allerdings doch mit erheblichem Aufwand
  - Ob dies sinnvoll ist kann hier weder entschieden noch gewertet werden
- Script-Sprachen wie JavaScript können mit Hilfe des Scripting APIs direkt innerhalb der Java Virtual Machine ausgeführt werden
- Weiterhin stehen mit Groovy und Scala andere Programmiersprachen zur Verfügung, die direkt Bytecode produzieren
  - Und damit einfach in Kombination mit Java benutzt werden können



## Code: Ein dynamisches Object

```
private void invoke(Object simpleObject, String
parameterlessMethodName) {
    try {
        Class<? extends Object> objectClass =
            simpleObject.getClass();
        Method methodToInvoke =
            objectClass.getMethod(parameterlessMethodNa
me);
        methodToInvoke.invoke(simpleObject);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



## Exkurs: Scripting API: Java

```
ScriptEngineManager manager = new  
ScriptEngineManager();  
ScriptEngine engine =  
manager.getEngineByName("nashorn");  
Reader r = new  
InputStreamReader(this.getClass().getResourceAsStream("script.js"));  
engine.eval(r);
```



## Exkurs: Scripting API: JavaScript

```
var company = {  
  name : "Javacream",  
  address : {  
    city : "Munich",  
    street : "Marienplatz"  
  },  
  info : function() {  
    return "The company " + this.name + " resides in  
    " + this.address.city  
  }  
}  
print(company.info())
```



## Exkurs: Groovy

```
class DynamicObject {  
    Object getProperty(String name) {  
        return 'it is a property!'  
    }  
    void setProperty(String name, Object value) {  
        println("setting property ${name} to ${value}")  
    }  
    def invokeMethod(String name, def args) {  
        println('executing invokeMethod')  
        return "OK"  
    }  
}
```





# Generics



Rekapitulation



Wildcards



Ein paar Details



Beispiel



# Rekapitulation



# Generische Collections

- Generische Datentypen wurden mit Java 1.5 im Collections-API eingeführt
  - `List<String> names = new ArrayList<String>();`
- Ab Java 7: Diamond-Syntax durch Type Inference
  - `List<String> names = new ArrayList<>();`
- Arbeitsweise:
  - Die generischen Datentypen werden vom Compiler durch automatisch hinzugefügte Casts realisiert
    - Damit könnte durch Reflection die generische Typisierung ausgehebelt werden
- Any
  - `List<?>` bedeutet, dass der Entwickler keinerlei Aussagen über die Typen der von der Liste verwalteten Datenbestand machen möchte
  - Damit sind alle `add`-Methoden nicht benutzbar
    - Die Liste ist quasi "read only"
  - Die Verwendung von Any reduziert sich in der Praxis damit auf die Deklaration von Parametern und Rückgabetypen



# Generics und Vererbung

- Die Vererbungshierarchie wird bei Generics "umgedreht":
  - So nicht:
    - `List<SchoolBook> schoolBooks = new ArrayList<>();`
    - `schoolBooks.add(new SchoolBook()); //OK`
    - `schoolBooks.add(new Book()); //Fehler`
  - Aber so:
    - `List<Book> books = new ArrayList<>();`
    - `books.add(new SchoolBook()); //OK`
    - `books.add(new Book()); //OK`



# Wildcards



## Any und Raw types

- Der Any-"Type" bedeutet, dass keinerlei Aussagen über den Typen benötigt werden
  - Eine so genannte "Unbound Wildcard"
    - `List<?> noType = new ArrayList<>();`
    - `Assert.assertEquals(0, noType.size()); // OK`
    - `// noType.add(new Object()); // compiler error`
- Raw types sind untypisierte Listen aus der Zeit vor Java 5
  - `List raw = new ArrayList<>();`
  - `raw.add(new SchoolBook());`
  - `raw.add(new Object());`
  - `raw.add(42);`
  - Identisch zu `List<Object>`



## Bounded Wildcards: Upper Bounded

- Upper Bounded: `? extends UpperType`
  - Jeder generische Typ ist zulässig, der den `UpperType` als Superklasse hat
  - Was bedeutet `List<? extends Book> books;`
    - `books = new ArrayList<Book>();`
    - `books = new ArrayList<SchoolBook>();`
    - `books = new ArrayList<Object>(); // compiler error`
    - `Book b = books.get(0);`
    - `books.add(b); // compiler error: could be SchoolBook or Book`

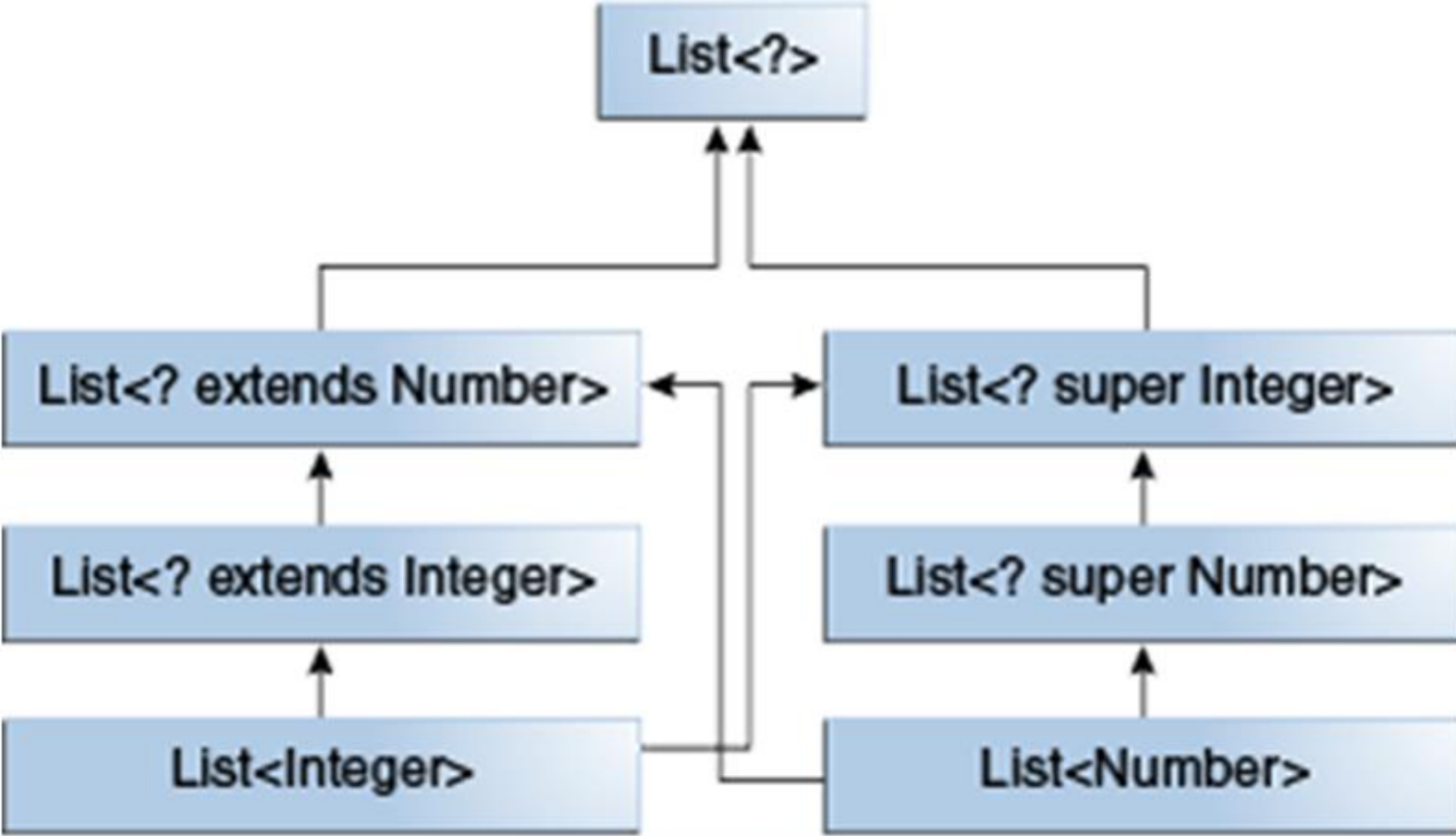


## Bounded Wildcards: Lower Bounded

- Lower Bounded: `? super LowerType`
  - Jeder generische Typ ist zulässig, Superklasse von `UpperType` ist
  - Was bedeutet `List<? super SchoolBook> books;`
    - `books = new ArrayList<Book>();`
    - `books = new ArrayList<SchoolBook>();`
    - `books = new ArrayList<SpecialistBook>(); //compiler error`
    - `books = new ArrayList<Object>();`
    - `Object o = books.get(0);`
    - `books.add(b);`
    - `books.add(new SchoolBook());`



# Wildcards und Vererbung



<http://docs.oracle.com/javase/tutorial/java/generics/subtyping.html>



## Ein paar Details



# Deklaration

- Generische Datentypen werden durch ein paar spitze Klammern deklariert
  - Mehrere Typen werden durch Komma getrennt
  - `<R, T>`
- Namenskonventionen
  - Type-Angabe besteht nur aus einem Großbuchstaben
  - Oracle-Konventionen
    - E - Element
    - K - Key
    - N - Number
    - T - Type
    - V - Value
    - S,U,V etc. - 2nd, 3rd, 4th types
- Deklaration
  - Klassendefinition
    - `class MyClass<T>`
  - Methodensignatur
    - `<T> String myMethod()`



# Generics sind keine Klassen

- Ein generischer Datentyp hat keinerlei Entsprechung zur Laufzeit
  - und wird damit nicht durch ein Klassenobjekt repräsentiert
    - Welches auch...
- Damit können auch keine der üblichen Klassenmethoden aufgerufen werden
  - Keine Instanziierung mit `new`
  - Kein Einstieg in das Reflection-API über `getClass()` möglich
- Auch ein Cast ist nur symbolisch und stellt den Compiler zufrieden
  - Allerdings gibt es immer noch eine Warnmeldung
    - `return (T) result`
    - Type safety: Unchecked cast from Object to T



# Generics und Reflection

- Type Erasure
  - Generic Types werden vom Compiler im Bytecode durch Casts ersetzt
    - Kein Overhead
    - Kein Erzeugen neuer Klassen
- Deshalb existieren zur Laufzeit Generics nicht



## Code: Generics und Reflection

```
ArrayList<String> list = new ArrayList<String>();  
list.getClass().getMethod("add", String.class) ;  
    //runtime error: method not found  
list.getClass().getMethod("add", Object.class) ;//OK
```



# Beispiel



## Ein Container für Benachrichtigungen

```
public class Notification<T> {  
    private T data;  
    public Notification(T data) {  
        super();  
        this.data = data;  
    }  
    public T getData() {  
        return data;  
    }  
    //...  
}
```





## Generischer TracingAspect

```
@SuppressWarnings("unchecked")
public static <T> T createAspects(T toDecorate) {
    ClassLoader classLoader =
        TracingAspect.class.getClassLoader();
    Class<?>[] interfacesToImplement =
        toDecorate.getClass().getInterfaces();
    TracingAspect tracingAspect = new TracingAspect();
    tracingAspect.setDelegate(toDecorate);
    return (T) Proxy.newProxyInstance(classLoader,
                                     interfacesToImplement,
                                     tracingAspect);
}
```



# Annotations



Rekapitulation



Eigene Annotationen



Annotations und Frameworks



Annotation Processors



# Rekapitulation



# Annotations als Meta-Informationen

- Mit der Einführung der Annotations wurde in Java ein typisiertes System für Meta-Informationen eingeführt
- Eine Annotation stellt sich hierbei dar als ein "Interface", das ausschließlich Zustand enthält
  - Kompletter Widerspruch zu den normalen Interfaces, die ausschließlich Verhalten enthalten können
  - Wie Interfaces können Annotations an jeder beliebigen Stelle im Rahmen eines Klassenmodells benutzt werden
    - Direkter Zusammenhang zu den UML-Stereotypen



# Annotation Processing

- Annotationen stellen die Informationen passiv zur Verfügung
- Ausgelesen werden diese
  - Durch Java Reflection
    - Das Reflection-API wurde dahingehend erweitert
      - z.B. `getClass().getAnnotations()`
  - Durch den Java Compiler
    - `@SuppressWarnings`
    - `@Override`
  - Durch einen Annotation Processor
    - Eine Art Precompiler
    - `Package javax.annotation.processing`



# Eigene Annotationen



# Target

- An welchen Stellen eine Annotation platziert werden darf wird bei der Deklaration durch das `Target` angegeben
- Gültige Werte sind:
  - `ANNOTATION_TYPE`
    - Annotation type declaration
  - `CONSTRUCTOR`
    - Constructor declaration
  - `FIELD`
    - Field declaration (includes enum constants)
  - `LOCAL_VARIABLE`
    - Local variable declaration
  - `METHOD`
    - Method declaration
  - `PACKAGE`
    - Package declaration
  - `PARAMETER`
    - Parameter declaration
  - `TYPE`
    - Class, interface (including annotation type), or enum declaration



# Retention

- Die Retention definiert, für welche Werkzeuge die Annotation zur Verfügung gestellt wird
  - oder etwas vereinfacht: Sollen die Annotationen vom Compiler in den Bytecode kompiliert werden?
- Zulässige Werte sind:
  - `CLASS`
    - Annotations are to be recorded in the class file by the compiler but need not be retained by the VM at run time
  - `RUNTIME`
    - Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively
  - `SOURCE`
    - Annotations are to be discarded by the compiler.





## Attribute von Annotationen

- Attribute von Annotationen sind read-only
- In der Annotations-Definition wird dafür eine parameterlose Zugriffsfunktion definiert
  - Vorsicht: Keine Bean-Property
    - also keine getter-Funktion
    - der Name des Attributs ist der Name der Funktion
- Es können default-Werte gesetzt werden



## Code: Eine einfache Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface SimpleJavaDocAnnotation {
    String description();
    String author() default "unknown";
}
```



# Annotations und Frameworks



# Übersicht

- Frameworks
  - verwalten Anwendungsklassen
  - stellen Container-Dienste zur Verfügung
  - müssen konfiguriert werden
- Die Konfigurationseinstellungen einer Anwendungsklasse werden sinnvoll durch Annotationen transportiert
  - Alternativen wie XML-Konfigurationsdateien oder ähnliches sind natürlich ebenfalls geeignet
    - Geschmackssache...
- Zum Auslesen von Annotationen wird das Reflection-API benutzt



## Code: Auslesen einer Annotation

```
public static <T> T createAspects(T toDecorate) {
    if (toDecorate.getClass().getAnnotation(Traced.class)
        != null) {
        ClassLoader classLoader =
            TracingAspect.class.getClassLoader();
        Class<?>[] interfacesToImplement =
            toDecorate.getClass().getInterfaces();
        TracingAspect tracingAspect = new TracingAspect();
        tracingAspect.setDelegate(toDecorate);
        return (T) Proxy.newProxyInstance(classLoader,
            interfacesToImplement, tracingAspect);
    }
    else {
        return toDecorate;
    }
}
```



# Annotations und Frameworks

- Praktisch alle modernen Java Frameworks definieren einen Satz geeigneter Annotationen
  - Web Frameworks
    - Servlets
    - Web Services
  - Context & Dependency Injection
  - Java Persistence API
  - ...



# Annotation Processors



## Processor-API

- Package `javax.annotation.processing`
- Einstiegs-Klasse `AbstractProcessor`
  - Lesen der Annotationen
  - Writer zum Schreiben generierten Codes





## javac-Konfiguration

- Durch Angabe der `processor` -Option wird die Prozessor-Klasse angegeben
  - Diese wird im `processorPath` oder im Klassenpfad gesucht
  - Alternativ: Standard-Such-Algorithmus
- Generierter Code wird automatisch mit kompiliert



# Ein simpler Prozessor

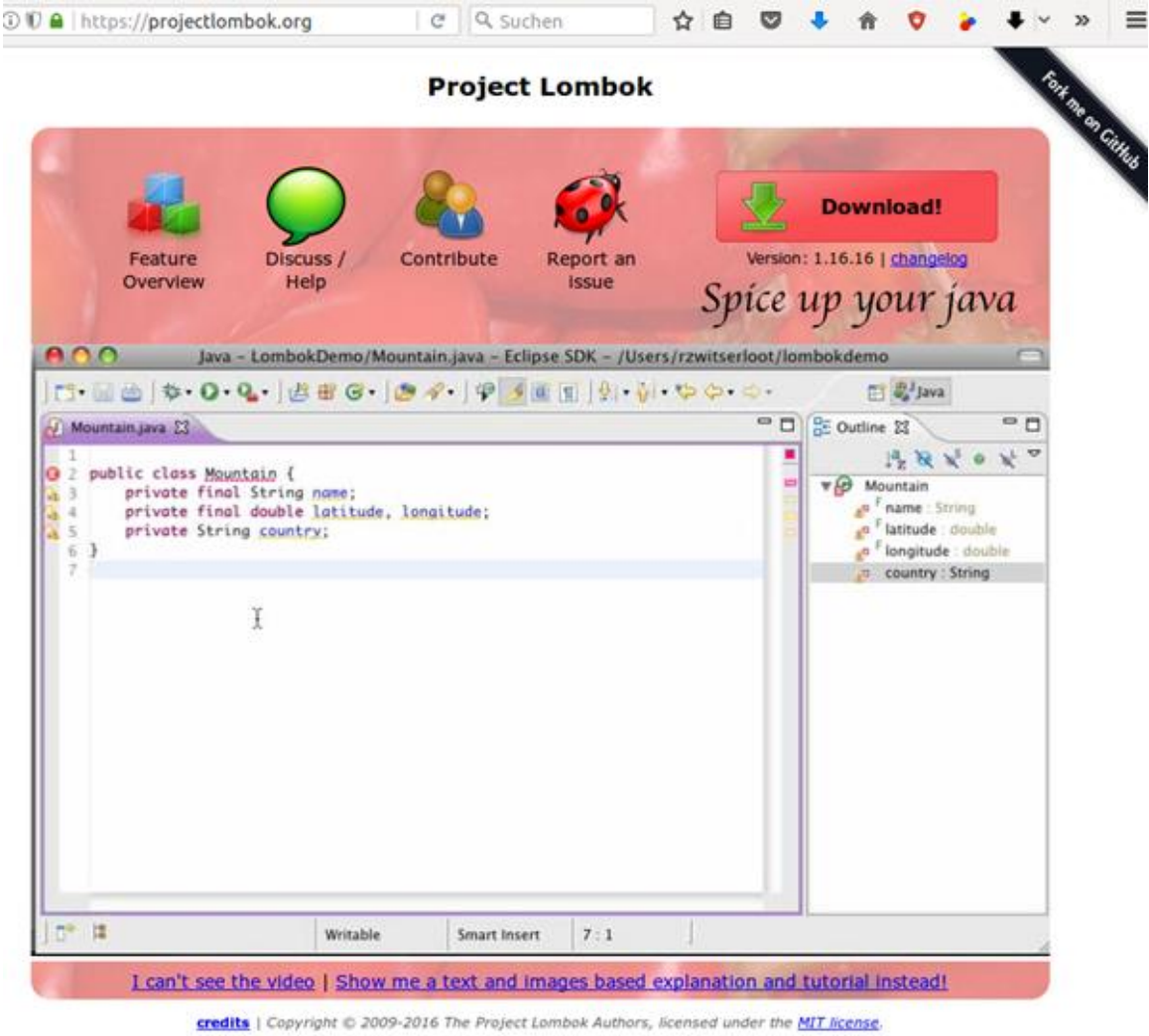
```
@SupportedAnnotationTypes("org.javacream.training.java.experts.annotations.TracingEnabled")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SimpleAnnotationProcessor extends AbstractProcessor {

    @Override
    public boolean process(Set<? extends TypeElement> typeElements, RoundEnvironment roundEnvironment) {
        if (typeElements.size() > 0) {
            Set<? extends Element> elements = roundEnvironment.getElementsAnnotatedWith(typeElements.iterator().next());
            Map<Boolean, List<Element>> annotatedMethods = elements.stream()
                .collect(Collectors.partitioningBy(element -> {
                    String simpleName = element.getSimpleName().toString();
                    return !(simpleName.startsWith("get") || simpleName.startsWith("set"));
                }));
            List<Element> validMethods = annotatedMethods.get(true);
            List<Element> invalidMethods = annotatedMethods.get(false);
            System.out.println("Valid methods: " + validMethods);
            System.out.println("Invalid methods: " + invalidMethods);

        } else {
            System.out.println("No matching annotation found");
        }
        return true;
    }
}
```



# Projekt Lombok





# Annotationen für Daten-Strukturen

- Triviale Methoden einer Daten-haltenden Klasse werden vom Lombok-Prozessor automatisch generiert
  - Konstruktoren
  - `hashCode` und `equals`
  - `toString`
  - getter und setter
- Damit ist das umständliche Erzeugen dieser Methoden über Wizards der IDE überflüssig
- Die Methoden werden direkt dem Bytecode zugeordnet, sind aber trotzdem in der Übersichtsdarstellung (Outline in Eclipse) sichtbar



# Eine Lombok-erweiterte Klasse

```
package org.javacream.training.java.experts.annotations;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class LombokPerson {

    private String name;
    private int height;
}
```

**LombokPerson**

- getName() : String
- getHeight() : int
- setName(String) : void
- setHeight(int) : void
- equals(Object) : boolean
- canEqual(Object) : boolean
- hashCode() : int
- toString() : String
- LombokPerson(String, int)
- LombokPerson()
- name : String
- height : int



# Funktionale Programmierung



Funktionsobjekte



Syntax



Funktionale Programmierung



# Funktionsobjekte



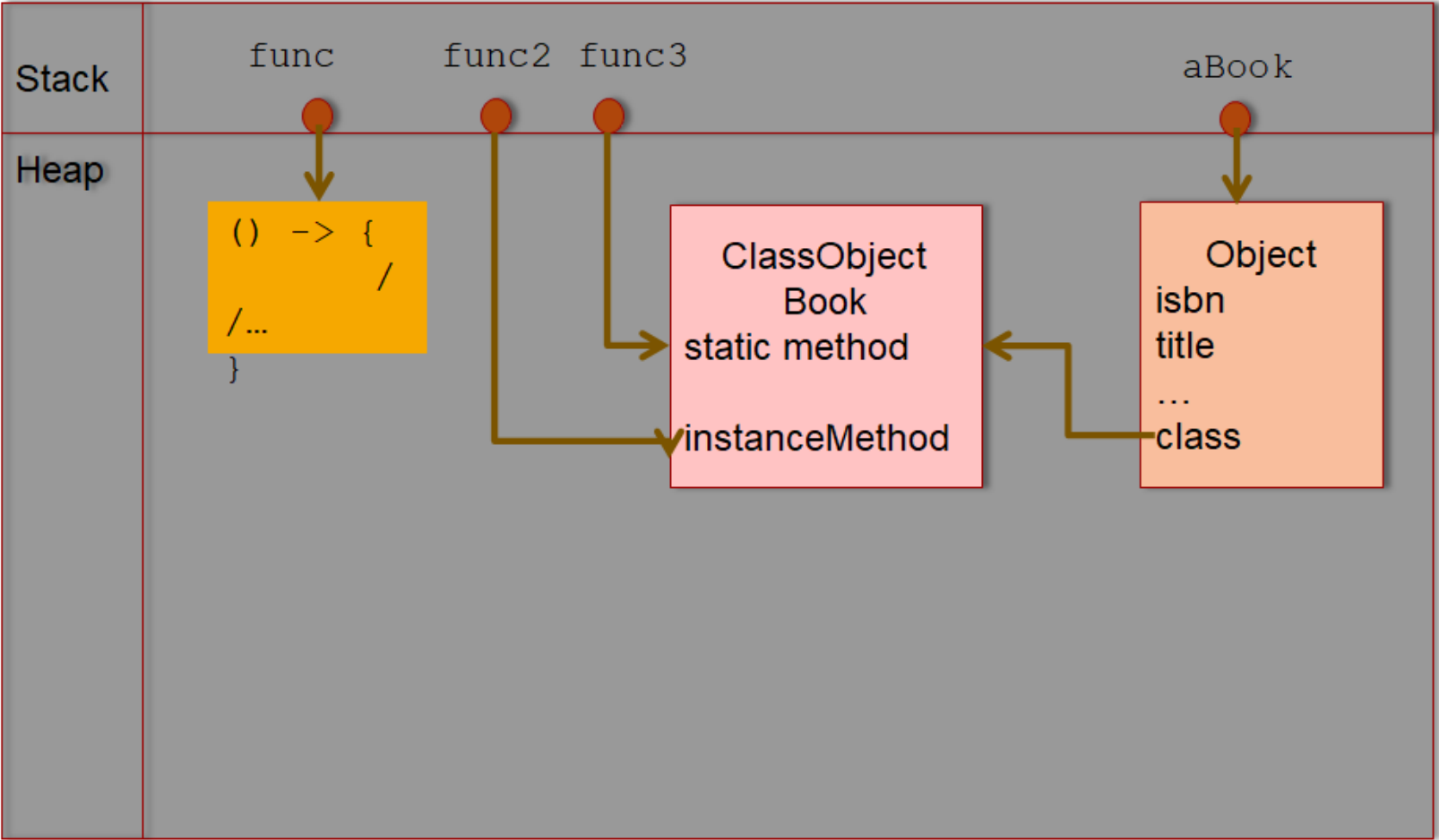
# Funktionen als Top-Level-Objekte

- Seit Java 8 existieren Funktionen als Top-Level-Objekte
  - Vorher mussten Algorithmen stets einer Methode einer Klasse zugeordnet werden
- Referenzen deuten auf
  - Funktionsobjekte im Heap
  - Methoden einer Klasse
    - Statische Methoden
    - Instanz-Methoden





# Klassenobjekte im Speicher der JVM





## Einordnung in das Java-Typsystem

- Nicht ganz einfach
- Trick: Einführen von "Functional Interfaces"
  - Diese deklarieren exakt eine abstrakte Methode
- Das Paket `java.util.function` deklariert eine Reihe gebräuchlicher funktionaler Interfaces
  - Damit wird eine Inflation eigener Deklarationen vermieden
- Wichtig: Jedes Interface mit nur einer einzigen abstrakten Methode ist ein funktionales Interface!
  - Die Annotation `java.lang.FunctionalInterface` ist nur ein Hinweis für den Java Compiler



# Funktionale Interfaces

Java™ Platform  
Standard Ed. 8

All Classes

Packages

java.applet  
java.awt  
java.awt.color  
java.awt.datatransfer  
java.io  
java.lang  
java.math  
java.net  
java.nio  
java.rmi  
java.security  
java.sql  
java.text  
java.time  
java.util  
java.util.concurrent  
java.util.function  
java.util.logging  
java.util.regex  
java.util.zip

java.util.function

Interfaces

BiConsumer  
BiFunction  
BinaryOperator  
BiPredicate  
BooleanSupplier  
Consumer  
DoubleBinaryOperator  
DoubleConsumer  
DoubleFunction  
DoublePredicate  
DoubleSupplier  
DoubleToIntFunction  
DoubleToLongFunction  
DoubleUnaryOperator  
Function  
IntBinaryOperator  
IntConsumer  
IntFunction  
IntPredicate  
IntSupplier  
IntToDoubleFunction  
IntToLongFunction  
IntUnaryOperator

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES

Package **java.util.function**

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
<b>BiConsumer&lt;T,U&gt;</b>	Represents an operation that accepts two input arguments and returns no result.
<b>BiFunction&lt;T,U,R&gt;</b>	Represents a function that accepts two arguments and produces a result.
<b>BinaryOperator&lt;T&gt;</b>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
<b>BiPredicate&lt;T,U&gt;</b>	Represents a predicate (boolean-valued function) of two arguments.
<b>BooleanSupplier</b>	Represents a supplier of boolean-valued results.
<b>Consumer&lt;T&gt;</b>	Represents an operation that accepts a single input argument and returns no result.
<b>DoubleBinaryOperator</b>	Represents an operation upon two double-valued operands and producing a double-valued result.
<b>DoubleConsumer</b>	Represents an operation that accepts a single double-valued argument and returns no result.
<b>DoubleFunction&lt;R&gt;</b>	Represents a function that accepts a double-valued argument and produces a result.
<b>DoublePredicate</b>	Represents a predicate (boolean-valued function) of one double-valued argument.



# Syntax



# Lambda-Ausdrücke: Das Funktions-Literal

- Allgemeine Form
  - `(Parameterliste) -> {Implementierung}`
- Parameterliste
  - Typisierte Komma-separierte Liste
- Implementierung
  - Beliebige Java-Sequenzen, die mit einem `return` beendet werden
- Verwendung ganz normal als
  - Parameter
  - Rückgabewert
  - Zuweisung an eine Variable
- Beispiel
  - ```
Function<String, String> func = (String message) -> {return "Hello " + message;};
```



# Syntactic Sugar

- Für Funktionslitterale kann Type Inference sehr nützlich benutzt werden
  - So kann auf die Typisierung der Parameterliste verzichtet werden
    - `Function<String, String> func = (message) -> {return "Hello " + message;};`
- Weiterhin ist die Block-Klammer und der Rückgabewert optional
  - `Function<String, String> func2 = (message) -> "Hello " + message;`
- Bei einem Parameter kann auch die runde Klammer weggelassen werden
  - `Function<String, String> func2 = message -> "Hello " + message;`



# Methoden-Referenzen

- `ContainingClass::staticMethodName`
  - Referenziert eine statische Methode
- `containingObject::instanceMethodName`
  - Instanz-Methode des Objekts
- `ClassName::new`
  - Konstruktor der angegebenen Klasse
- `ContainingType::methodName`
  - Instanz-Methode eines zufällig gewählten Objekts



## Einige Details

- Funktionsobjekte und Klassen
  - Im Gegensatz zu Klassendeklarationen werden Funktionsobjekte nicht zu einer Bytecode-Datei kompiliert
    - Damit sind diese weniger aufwändig als beispielsweise anonyme Klassen
- Typisierung
  - Der Typ eines Funktions-Literals wird durch ausgefeiltes "Target Typing" bestimmt
    - Je nachdem, wo der Lambda-Ausdruck definiert wird kann er völlig unterschiedliche Typen aufweisen





## Code: Type Inference bei Lambda- Ausdrücken

```
public void targetTypeing() {  
    doSomethingWithInteger(p -> p + 1); //43  
    doSomethingWithString(p -> p +  
1); // "421"  
}
```

```
private void doSomethingWithInteger(  
    Function<Integer, Integer> callback) {  
    System.out.println(callback.apply(42));  
}  
private void doSomethingWithString(  
    Function<String, String> callback) {  
  
    System.out.println(callback.apply("42"));  
}
```



# Closures

- Der Begriff "lokale Variable" muss bei der Verwendung von Lambda-Ausdrücken aufgegeben werden
  - Das Funktionsobjekt hat Zugriff auf alle lokalen Variablen des Kontextes, in dem sie definiert wurde
  - Damit verlängert sich die Lebensdauer der lokalen Variable auf die Lebensdauer des Funktionsobjekts
  - Das ist der "Closure"-Effekt
    - Gilt übrigens ab Java 8 auch für Klassendefinitionen innerhalb einer Methode, also insbesondere für Anonyme Klassen
- Hinweis:
  - Die innerhalb einer Closure verwendeten Variablen des äußeren Kontextes müssen "effektiv final" sein
    - Damit sind Closures in Java 8 noch nicht vollständig umgesetzt



## Code: Beispiel einer Closure

```
public void closures() {  
    int increment = 2;  
    Integer[] values = {5, 6};  
    storeLambda(p -> p + values[1] + increment);  
    System.out.println(func.apply("47"));  
    //increment = 3; //compiler error:
```

in Local variable increment defined  
  
or an enclosing scope must be final  
  
effectively final

```
    values[1] = -11;  
    System.out.println(func.apply("47"));  
}  
private void storeLambda(  
    Function<String, String> callback){  
    FunctionalTests.func = callback;  
}
```



# Funktionale Programmierung



# Einführung

- Mit Java hat sich in den letzten 20 Jahren das Paradigma der objektorientierten Programmierung als Standard etabliert
- Alternativ dazu wurde am Paradigma der funktionalen Programmierung gearbeitet
  - mit Sprachen wie Lisp, ML und anderen
- Auch Java wurde um funktionale Elemente erweitert
  - wie auch C# und C++

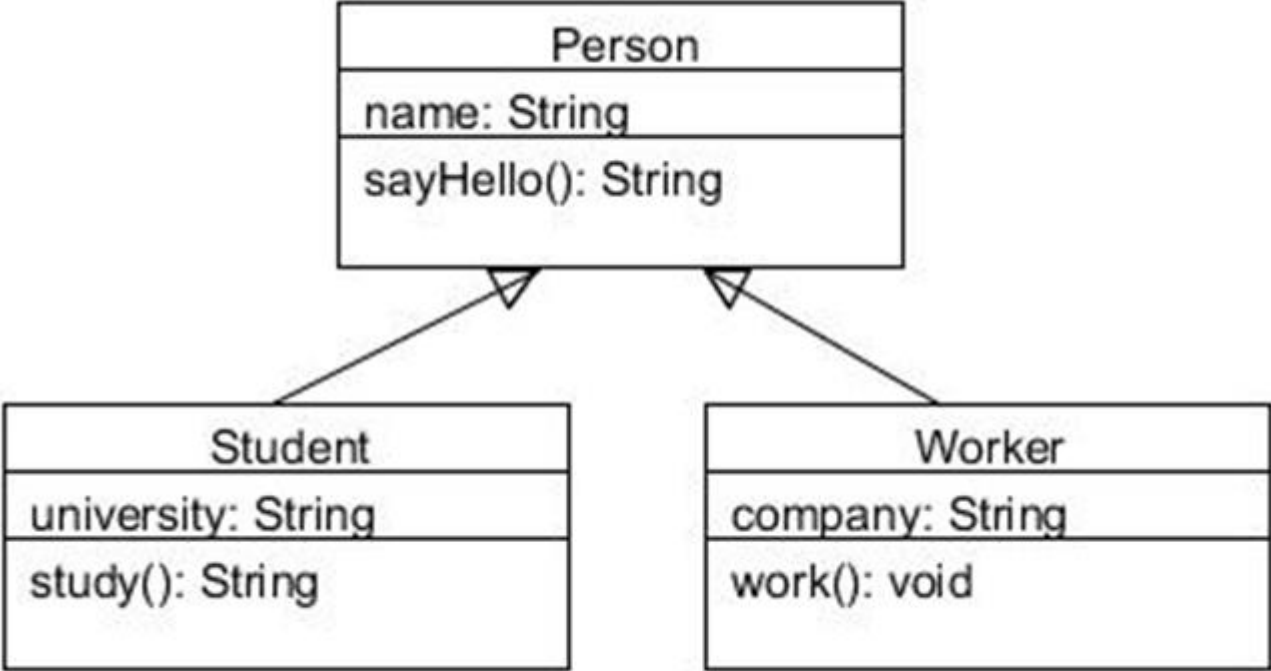


# Funktionale Ansätze

- Funktionale Sprachen beschreiben das „Was“
  - OOP fokussiert auf das „Wie“
- Funktionale Grundannahmen:
  - Funktionen werden als Werte behandelt
  - Die Typen von Funktionen werden automatisch vom Compiler bestimmt
  - Die funktionale Programmierung favorisiert immutable Elemente
    - Diese lassen sich beispielsweise einfacher verstehen als änderbare Elemente und bedürfen keiner Synchronisation
  - Durch die partielle Auswertung (Currying) von Funktionen können Parameter reduziert werden um definierte Berechnungen auszuführen



# Ein einfaches Klassendiagramm





## Diskussion

- Auf den ersten Blick doch ein schönes Beispiel für eine Objekt-orientierte Modellierung
- Aber!
  - Was passiert, wenn wir einen arbeitenden Studenten benötigen?
  - Oder einen studierenden Arbeiter?
  - Oder noch schlimmer: Aus einem bereits erzeugten Studenten soll ein Arbeiter werden oder umgekehrt?





# Java und XML



Parser



XML Transformationen



XML Serialisierung



Object/XML-Mapping



Grundlagen der Client-Server-Programmierung



# Parser



# Überblick Parser

- Arbeitsweise von Parsern ist unabhängig von Java spezifiziert
  - W3W-Konsortium
  - xml.org
- Verschieden Modelle
  - SAX: Simple API for XML
    - Der SAX-Parser verarbeitet das gesamte Dokument und feuert Events
      - Push-Mechanismus
    - Verarbeitung durch eigene Listener
  - DOM: Das Document Object Model
    - Der DOM-Parser transformiert das komplette XML-Dokument in einen Graphen von Objekten
    - Verarbeitung durch DOM-Operationen
      - Navigieren
      - Selektieren
      - Transformieren
  - StAX: Streaming API for XML
    - Der Parser wird über die Anwendung gesteuert
      - Ein Pull-Mechanismus



## JAXP: Das Java API for XML Processing

- Einstiegspunkt in die Java-basierte XML-Verarbeitung
  - Package `javax.xml.parsers`
- Ab hier Unterteilung in die einzelnen Technologien
  - `org.xml.sax`
  - `org.w3c.dom`
  - `javax.xml.stream`



## Die dunkle Seite von Java und XML

- Die Java-APIs sind bei den Anwendungsprogrammierern nicht sonderlich beliebt
  - Kritikpunkte
    - Recht komplex
    - Untypisiert
  - Große Zahl von Framework-Lösungen ist vorhanden
    - JDOM
    - Apache XML Project
- Die zu verwendenden Parser-Implementierungen können im Programm definiert werden
  - System-Properties, z.B.  
`javax.xml.parsers.DocumentBuilderFactory`
  - Problem
    - Die Implementierungen benutzen aus Effizienz-Gründen teilweise undokumentierte Java-Features, die selbst bei minimalen Versions-Upgrades geändert werden
      - `ClassCastException`
      - `NoSuchMethodError`
    - Bei älteren Java-Versionen unglaublich aufwändig



# XML

## Transformationen



## XSL und JAXP

- Bestandteil der Extensible Stylesheet Language (XSL)
  - XSLT
    - Transformation
  - XPath
    - Selektion
  - XSL-FO
    - Formatting Objects
- JAXP
  - Package `javax.xml.transform`
    - Mit den Subpaketen `sax`, `dom`, `stream`
  - Package `javax.xml.xpath`
  - XSL-FO wird nicht von JAXP unterstützt
    - Apache FOP-Framework
    - Reporting-Produkte
      - Jasper
      - Eclipse BIRT



# XSL-Übersicht

The screenshot shows the 'XSLT Introduction' page on w3schools.com. The browser address bar displays 'https://www.w3schools.com/xml/xsl\_intro.asp'. The page has a dark navigation bar with links to HTML, CSS, JAVASCRIPT, SQL, PHP, BOOTSTRAP, and MORE. A sidebar on the left lists various XML and XSL topics. The main content area is titled 'XSLT Introduction' and includes a green box with the following text:

XSL (EXtensible Stylesheet Language) is a styling language for XML.

XSLT stands for XSL Transformations.

This tutorial will teach you how to use XSLT to transform XML documents into other formats (like transforming XML into HTML).

Below this, there is a section for an 'Online XSLT Editor' with the text: 'With our online editor, you can edit XML and XSLT code, and click on a button to view the result.' Underneath, an 'XSLT Example' section shows the XML declaration code: `<?xml version='1.0'?>`.

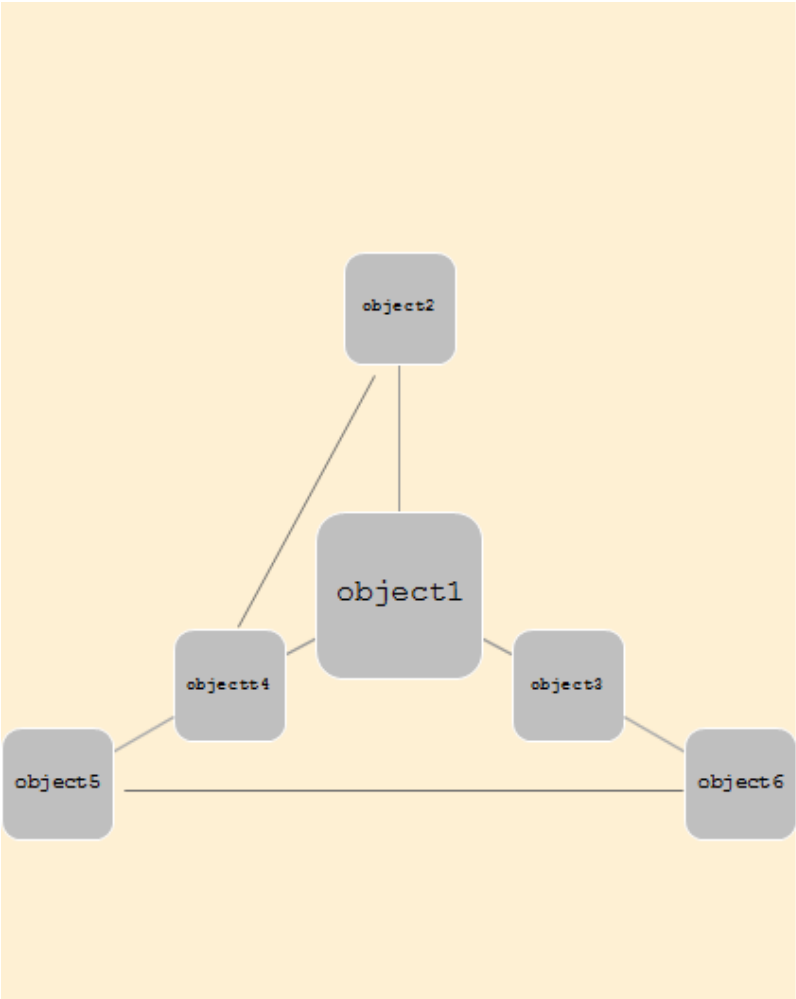
The right sidebar contains a Google AdWords advertisement with the text: 'Fatti notare con Google AdWords.' and a blue button labeled 'COMINCIA ORA'. Below the button, it says 'Con un credito di €75\*'. The Google AdWords logo is visible at the bottom of the sidebar.





# XML Serialisierung

# Serialisierung



File

```
010010111010100010
100101010001001001
010101001001010010
101010010100011101
011111100101101010
010100100101000011
```



## XML-Serialisierung

- Umwandlung des Objekt-Graphen in ein hierarchisches XML-Dokument
- Standard-Implementierung in `java.beans`
  - `XMLEncoder`
  - `XMLDecoder`
- Das generierte XML-Dokument unterscheidet Properties von Methoden
  - So werden die Elemente einer `java.util.ArrayList` durch `add-`Methodenaufrufe hinzugefügt
    - Das interne Array wird nicht serialisiert!



## Code: Eine serialisierte ArrayList

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_121" class="java.beans.XMLDecoder">
  <object class="java.util.ArrayList">
    <void method="add">
      <object
class="org.javacream.training.java.experts.xml.Book">
        <void property="isbn">
          <string>ISBN1</string>
        </void>
        <void property="price">
          <double>19.99</double>
        </void>
        <void property="title">
          <string>Java</string>
        </void>
      </object>
    </void>
  </java>
```



# Code: Ein Objektgraph mit zirkularen Referenzen

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_121" class="java.beans.XMLDecoder">
  <object
class="org.javacream.training.java.experts.xml.serialization.Class1"
id="Class10">
    <void property="class2">
      <object
class="org.javacream.training.java.experts.xml.serialization.Class2">
        <void property="class1">
          <object idref="Class10"/>
        </void>
        <void property="description">
          <string>class2 description</string>
        </void>
      </object>
    </void>
    <void property="description">
      <string>class1 description</string>
    </void>
  </object>
</java>
```



# Object/XML- Mapping



# Einführung

- JAXB ist das Java Framework für XML Binding
  - SAX, DOM und StAX sind allgemeine Parser-Lösungen
  - They will parse any well-structured XML
- JAXB liest erzeugt für jedes Schema einen speziellen Parser
- Analog zum DOM wird ein Objekt-Graph angelegt
  - Dieser ist jedoch typsicher
- Ebenso in der anderen Richtung
- Dies ist das O/X-Mapping



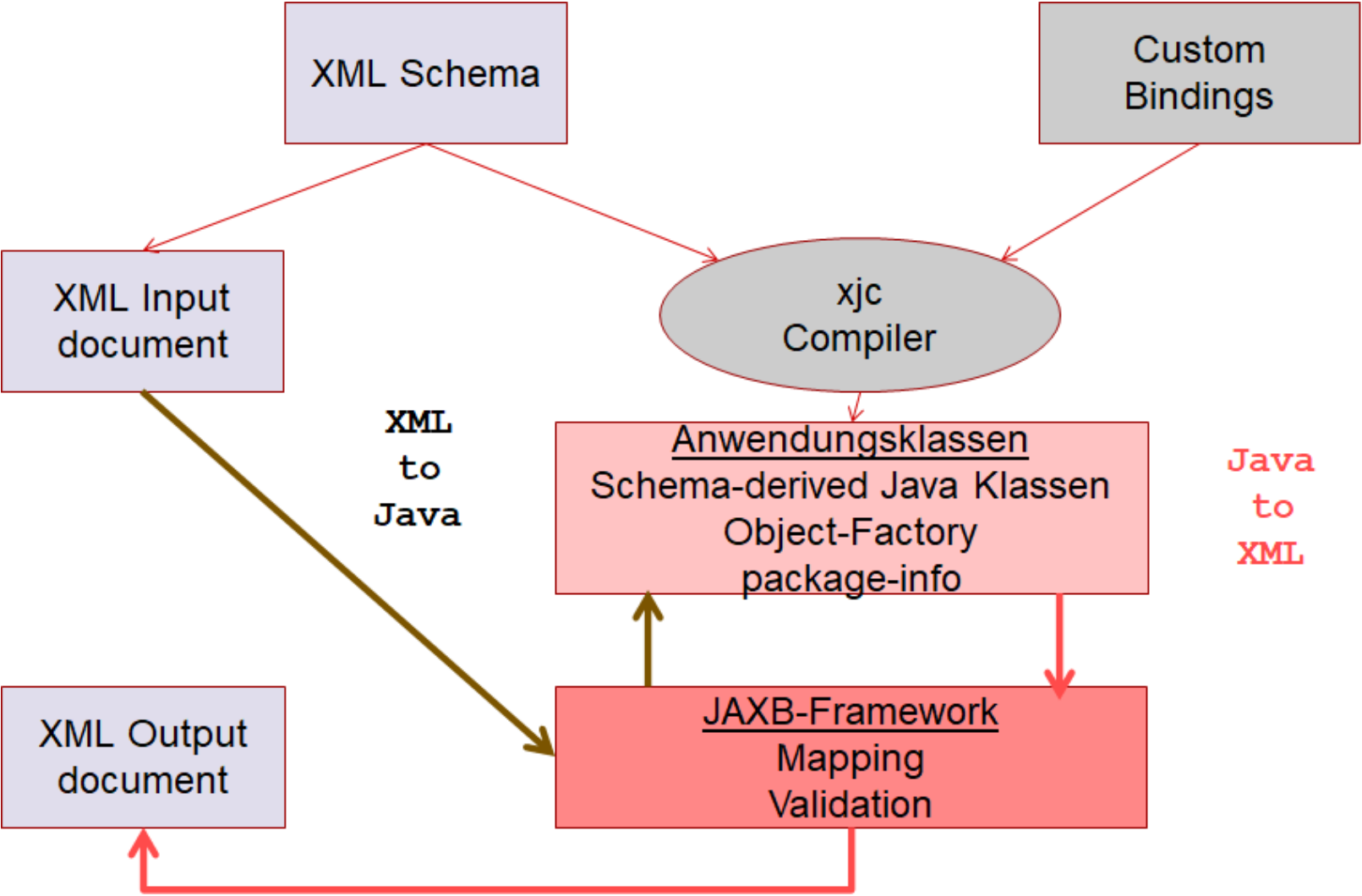
## Arbeitsweise

- JAXB benutzt Annotations zur Definition der Mapping-Informationen
  - Package `javax.xml.bind.annotation`
- Das Framework stellt einen Code-Generator zur Verfügung
  - Der Xml-Java-Compiler `xjc`
    - Sonst wären die annotierten Klassen auch recht unhandlich zu erstellen
  - Integration in Entwicklungsumgebungen ist gegeben





# Architektur

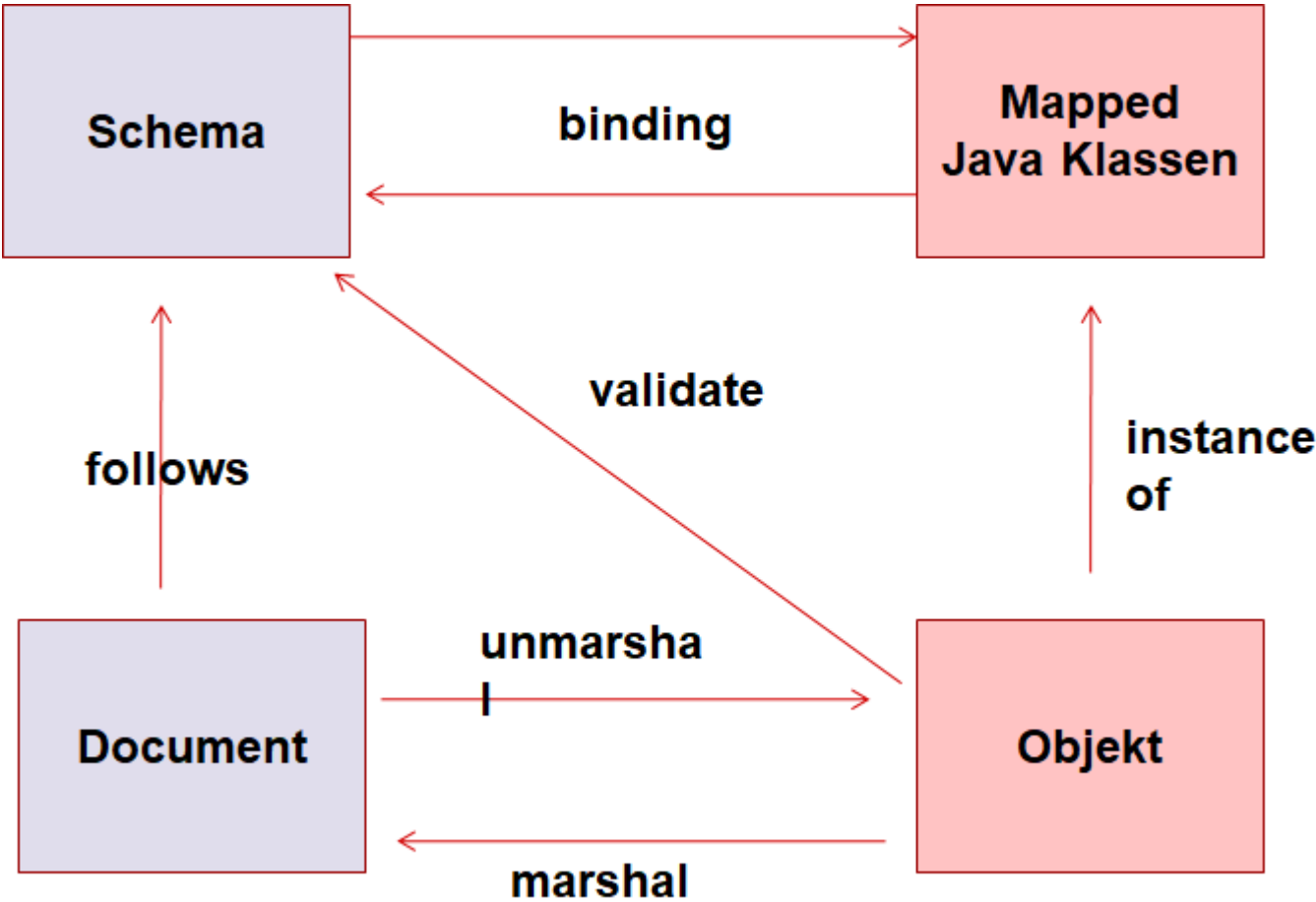




# Bindings

- Überschreiben der "reasonable defaults" des xjc-Compilers
- Inline als Bestandteil des XML-Schemas
  - `<xs:annotation>`
  - `<xs:appinfo>`
  - binding declarations
  - `</xs:appinfo>`
  - `</xs:annotation>`
- Externe Bindings-Datei
  - Endung .xjb
  - Format:
    - `<jxb:bindings schemaLocation = "xs:anyURI">`
    - `<jxb:bindings node = "xs:string">*`
    - `<binding declaration>`
    - `</jxb:bindings>`
    - `</jxb:bindings>`
    -

# Zusammenhänge





## Code: JAXB Unmarshalling

```
JAXBContext jaxbContext =  
JAXBContext.newInstance(  
    this.getClass().getPackage().getName());  
Unmarshaller unmarshaller =  
    jaxbContext.createUnmarshaller();  
JAXBElement<PublisherType> element = unmarshaller  
    .unmarshal(new StreamSource(  
        getClass().getResourceAsStream(xmlFilename)  
    ), PublisherType.class);  
PublisherType publisherType = element.getValue();
```



## Code: JAXB Marshalling

```
PublisherType publisherType = new PublisherType();
publisherType.setCity("Heidelberg");
publisherType.setName("Springer");
for (int i = 0; i < 2; i++) {
    BookType bookType = new BookType();
    bookType.setIsbn("ISBN" + i);
    bookType.setTitle("Title" + i);
    bookType.setPrice(9.99 + i);
    publisherType.getBook().add(bookType);
}
JAXBContext jaxbContext =
JAXBContext.newInstance(this.getClass().getPackage().get
tName());
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.marshal(
    new
ObjectFactory().createPublisher(publisherType),
    System.out);
```



## Code: JAXB Validierung

```
JAXBContext jaxbContext =
JAXBContext.newInstance(
    this.getClass().getPackage().getName());
SchemaFactory schemaFactory =
    SchemaFactory.newInstance(
        XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema(
    new StreamSource(
        getClass().getResourceAsStream(xsdFilename)
    ));
Marshaller marshaller =
    jaxbContext.createMarshaller();
marshaller.setSchema(schema);
```



## Alternativen zu JAXB

- MOXy
  - Alternative JAXB-Implementierung des EclipseLink-Projekts
- JiBX
- Apache Digester



# Grundlagen der Client-Server- Programmierung





# Networking mit Java

- Datenaustausch über Netzwerk ist Stream-basiert
  - Package `java.io`, `java.nio`
- Netzwerk-Protokolle
  - Package `java.net`
    - TCP/IP-Sockets
    - UDP
    - http
- Remote Method Invocation
  - Objektorientiertes Protokoll zwischen zwei Java Virtual Machines
  - Package `java.rmi`
- JDBC
  - Kommunikation mit Datenbank-Systemen
    - ausgerichtet auf relationale Datenbanken



# Höherwertige Protokolle

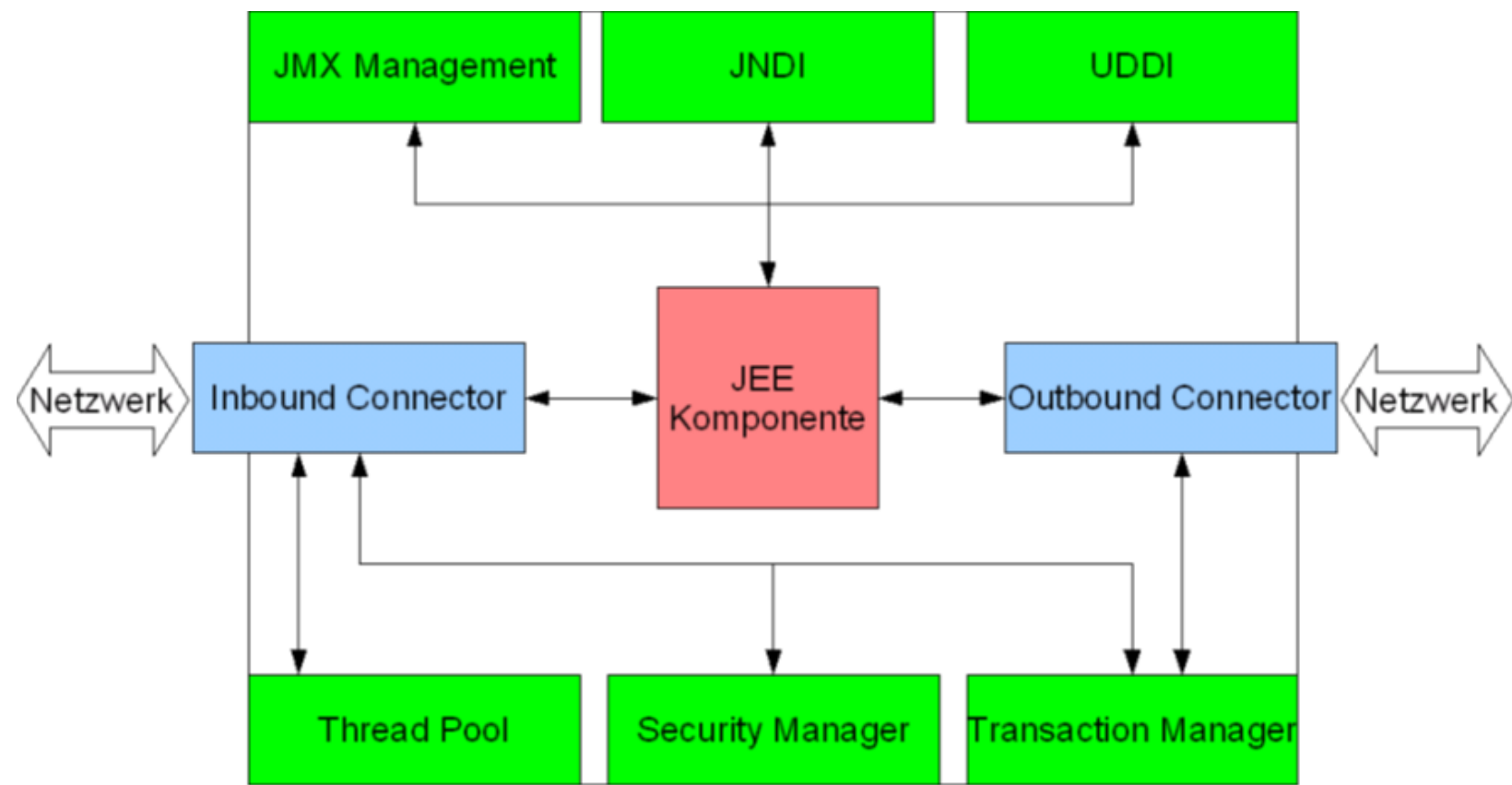
- http-basiert
  - Servlet-Technologie
    - Low level
  - Web Services
    - SOAP-basiert
      - JAX-WS
    - RESTful
      - JAX-RS
  - Browser-basierte Anwendungen
    - JavaServer Faces, JSF
- Messaging
  - Java Message Service, JMS
    - Bestandteil der Java Enterprise Edition
- Überwachung und Monitoring
  - Java Management Extension, JMX



## Java Server

- Eigene Implementierungen sind möglich, aber aufwändig
  - Umsetzen des Kommunikations-Protokolls
  - Multithreading für parallele Zugriffe
  - Installation und Aktualisierung von Anwendungen im laufenden Betrieb
    - "Hot Deployment"
  - Überwachung
- Dafür gibt es seit langem fertige und ausgereifte Lösungen

# Der Applikationsserver





# Einführung in das Java Persistence API



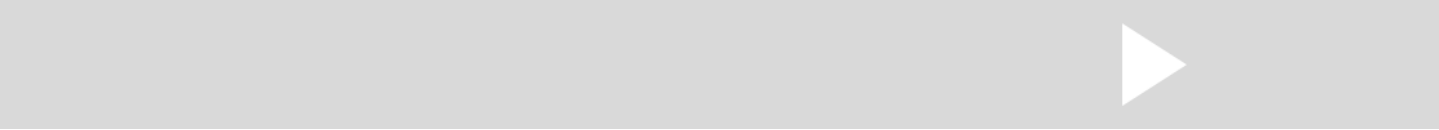
Überblick



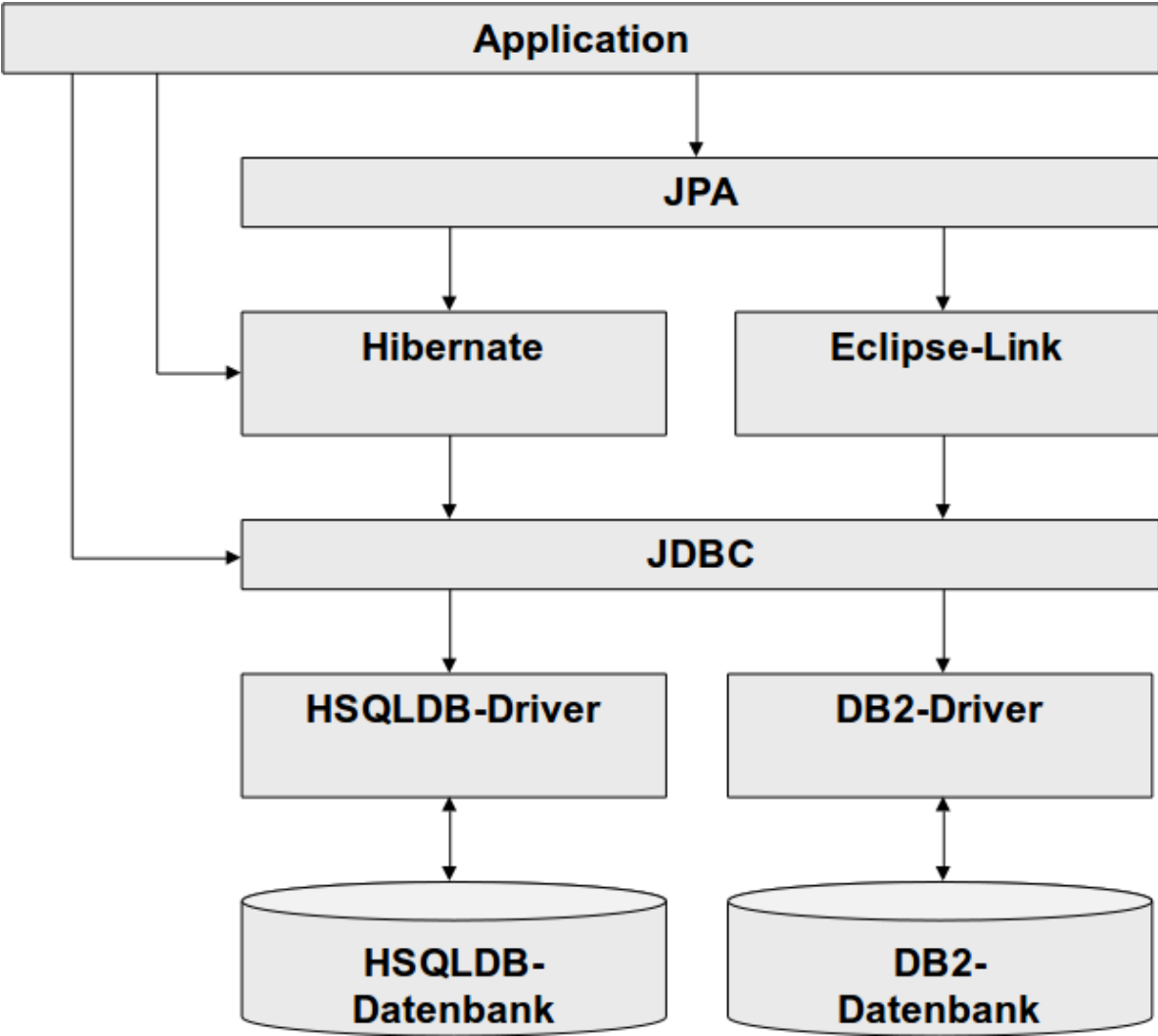
Der EntityManager



# Überblick

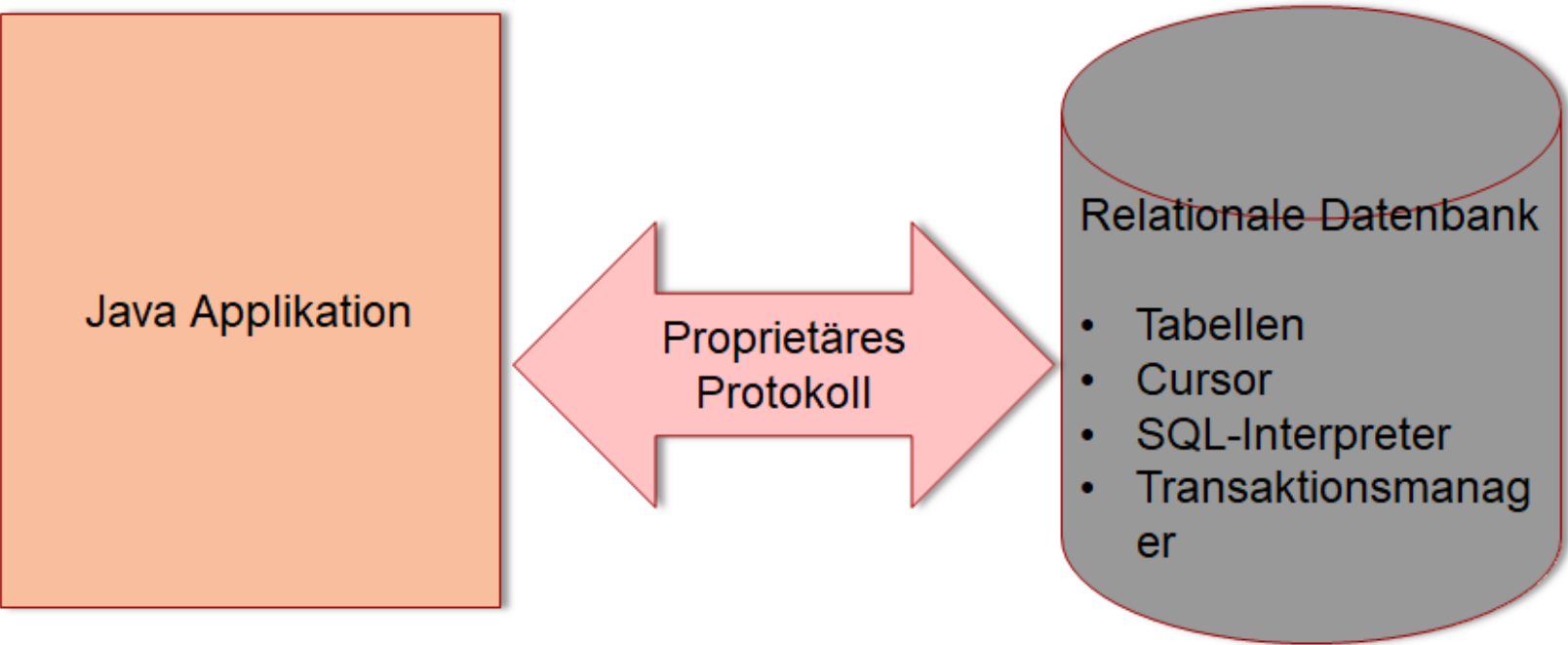


# Zusammenhänge

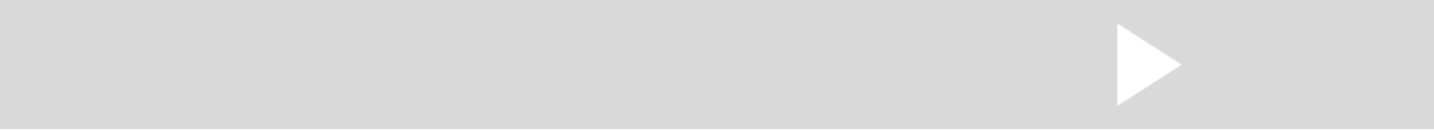
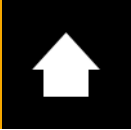




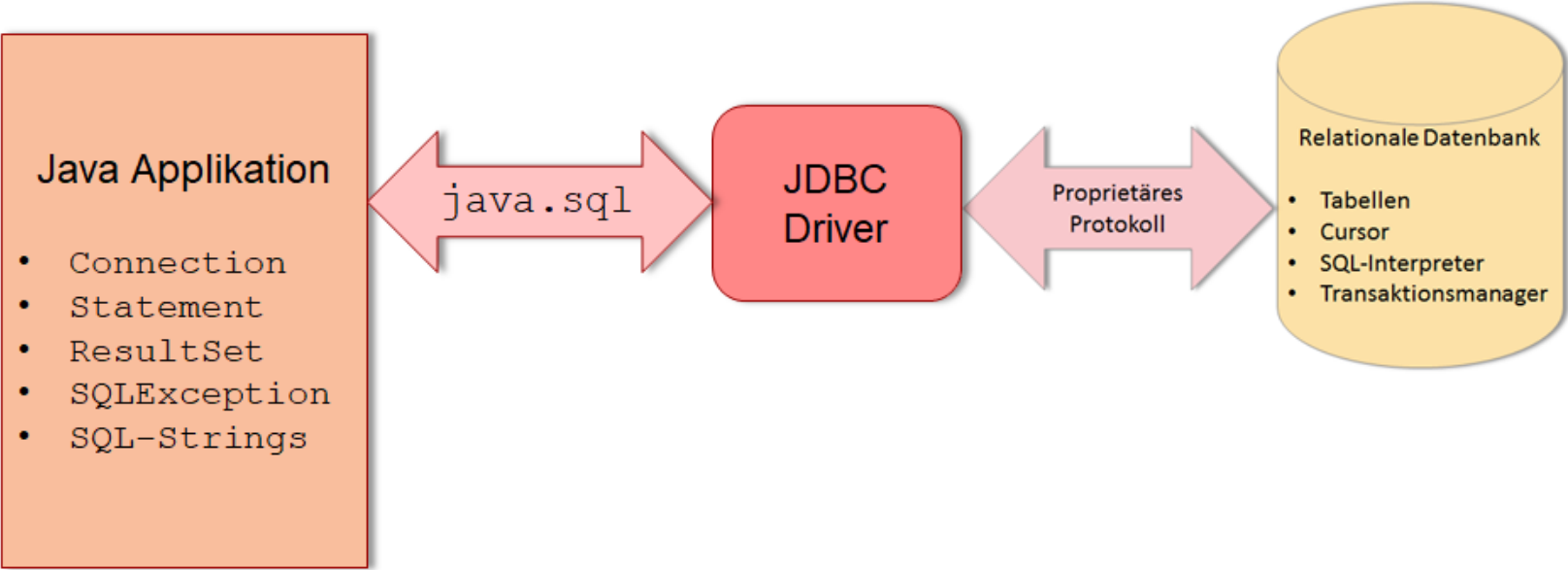
# Direkter Datenbankzugriff

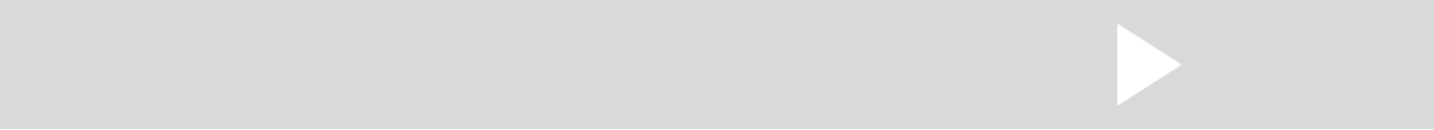




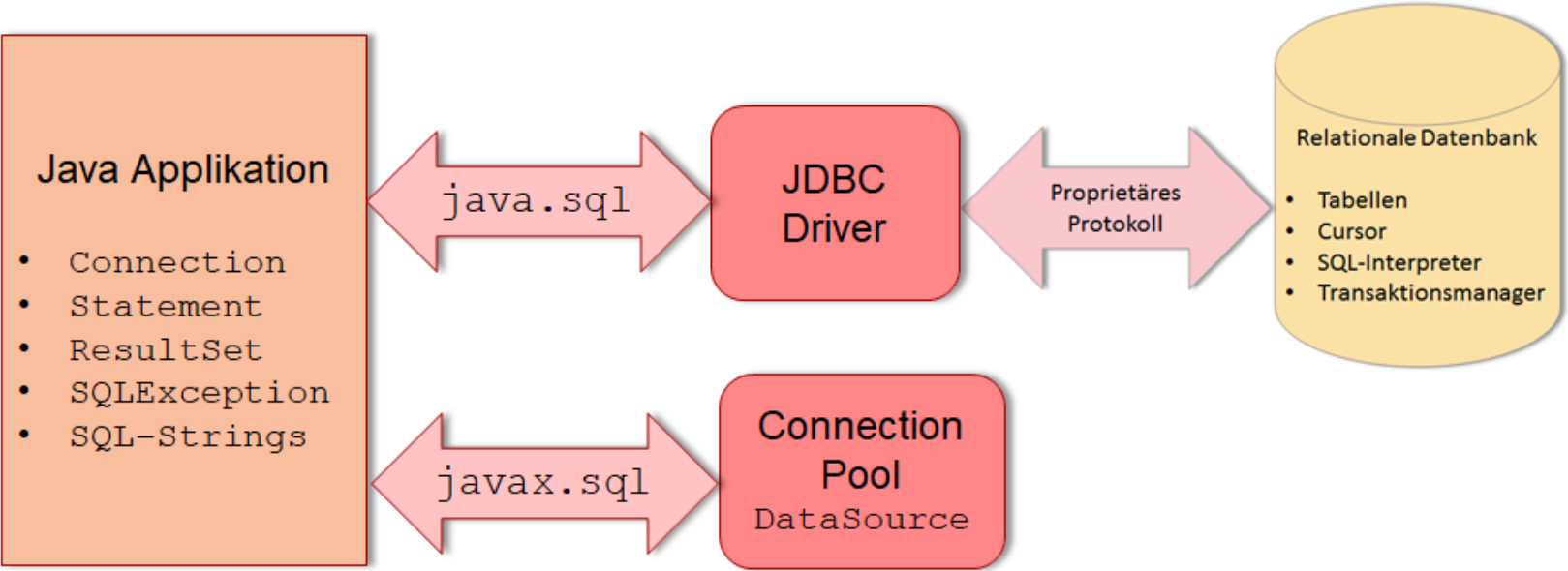


# JDBC Driver



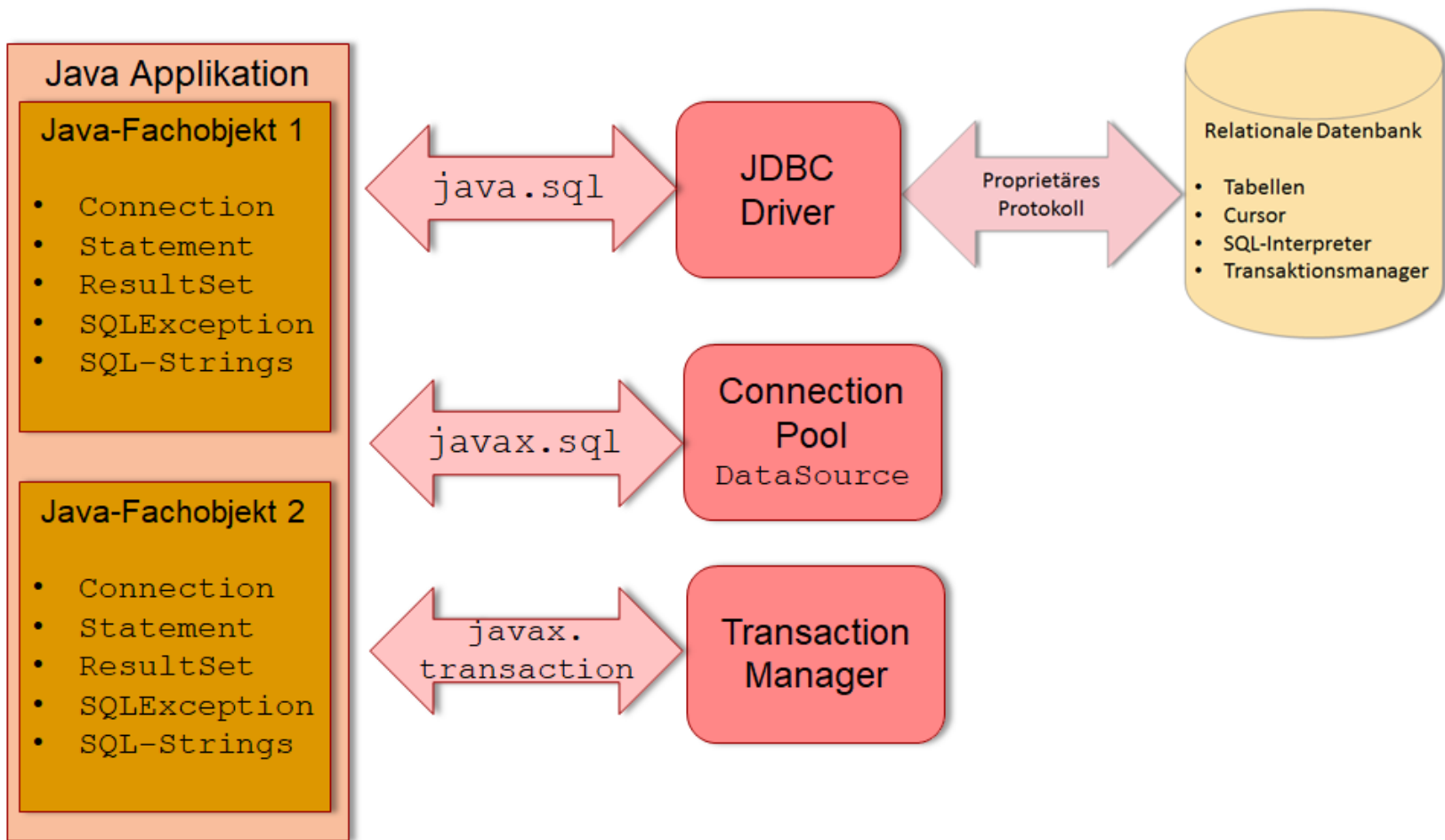


# Connection Pool





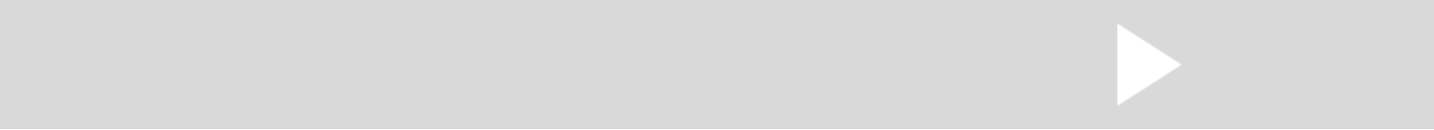
# Transaction Manager



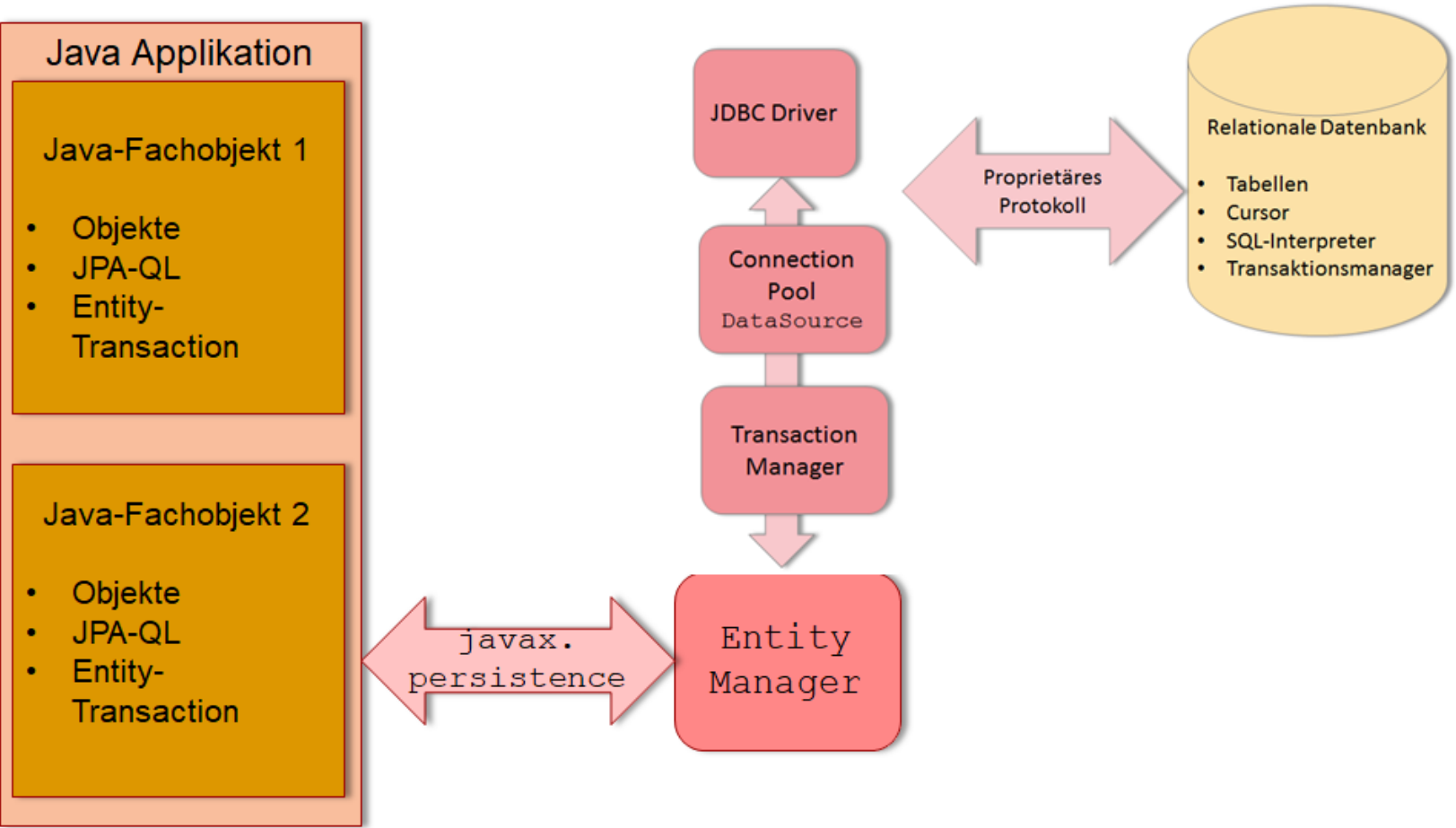


## Ablauf

- Die Fachklassen holen sich von der `DataSource` gepoolte Connections
- Der Transaction Manager koordiniert bei Bedarf einen gemeinsamen Transaktionskontext
  - So werden `commit` und `rollback`-Operationen über Aufrufe der Fachobjekte hinweg konsistent ausgeführt
- Die Kommunikation mit der Datenbank erfolgt jedoch weiterhin über SQL-Strings
- Insgesamt relativ kompliziert
  - "historisch gewachsen"



# Transaction Manager





# Der EntityManager



# Der EntityManager

- Einheitliches und konsistentes API für den Datenbankzugriff
- OOP-Konzepte
  - Datenbank-Identität über den Primärschlüssel wird zur Objekt-Identität über Referenzen
    - First Level Cache
  - O/R-Mapping
    - Annotations oder XML-Konfiguration
  - `Criteria`-Objekte für Suchen
- Daneben aber auch ein SQL-ähnliches Konzept mit JPA-QL
  - Native Queries



# Context & Dependency Injection



Überblick und Arbeitsweise



Provider und Beispiel





# Überblick und Arbeitsweise



# Überblick

- Context and Dependency Injection oder kurz CDI ist in modernen Anwendungen die Grundlage des Anwendungsdesigns
- Als zentrale konkrete Komponente von CDI existiert die Context-Implementierung
  - Diese wird in der Regel nicht selbst entwickelt sondern über ein Framework zur Verfügung gestellt
- Aktuelle existieren zwei relevante Implementierungen
  - Spring Framework
  - CDI-Framework als Bestandteil der Java Enterprise Edition



# Arbeitsweise des Context: Instanziierung



# Erzeugen der Objekte



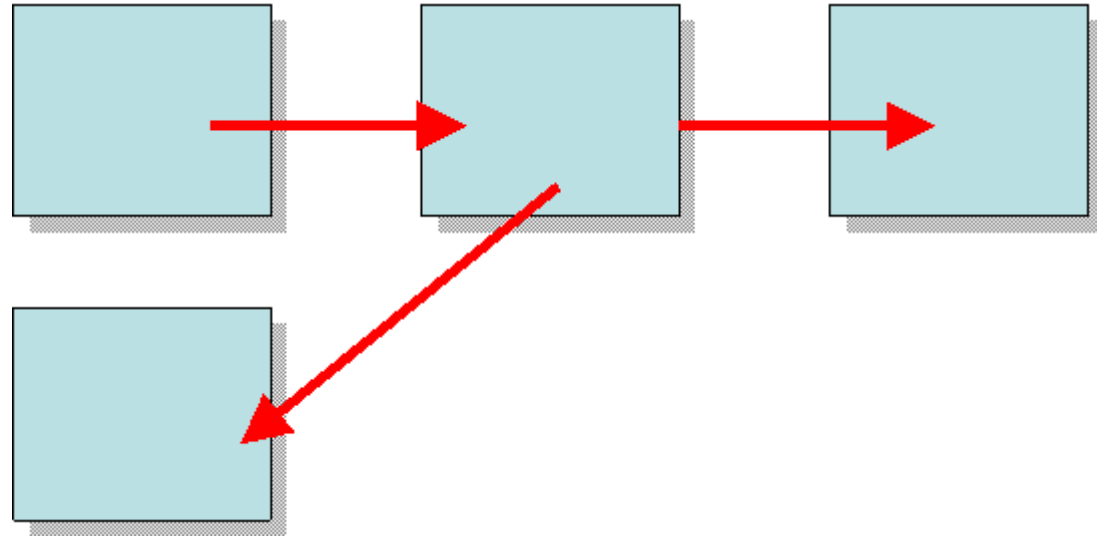


## Arbeitsweise des Context: Setzen der Dependencies

- Benutze Dependency Injection, um Abhängigkeiten zwischen den Objekten aufzubauen
- Zur Identifikation der Dependencies werden ebenfalls Meta-Daten benutzt
- Damit wird aus den singulären Objekten ein komplexes Objekt-Geflecht



# Dependency Injection





# Arbeitsweise des Context: Zugriff auf die Objekte



# Provider und Beispiel





## CDI-Provider

- Aktuell zwei relevante Provider
  - Spring
  - CDI der Java Enterprise Edition
- Spring ist neben dem reinen CDI-Framwork auch eine Sammlung nützlicher Utilities
- CDI ist eine Spezifikation, die von verschiedenen Providern unterstützt wird
  - JBoss Weld als Referenz-Implementierung
  - Jeder Applikationsserver mit JEE-Version  $\geq 6$
  - Auch Spring unterstützt CDI rudimentär