

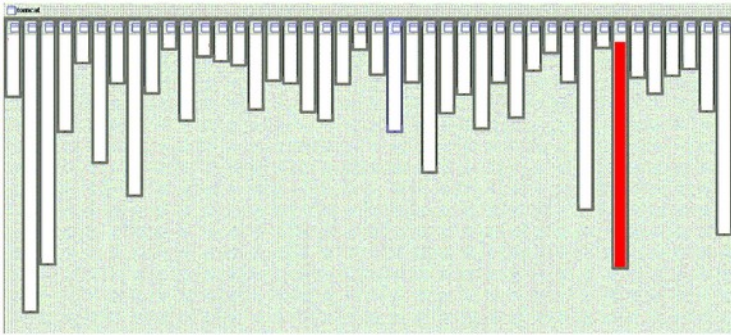
JAVA

Aspects (AOP) AspectJ

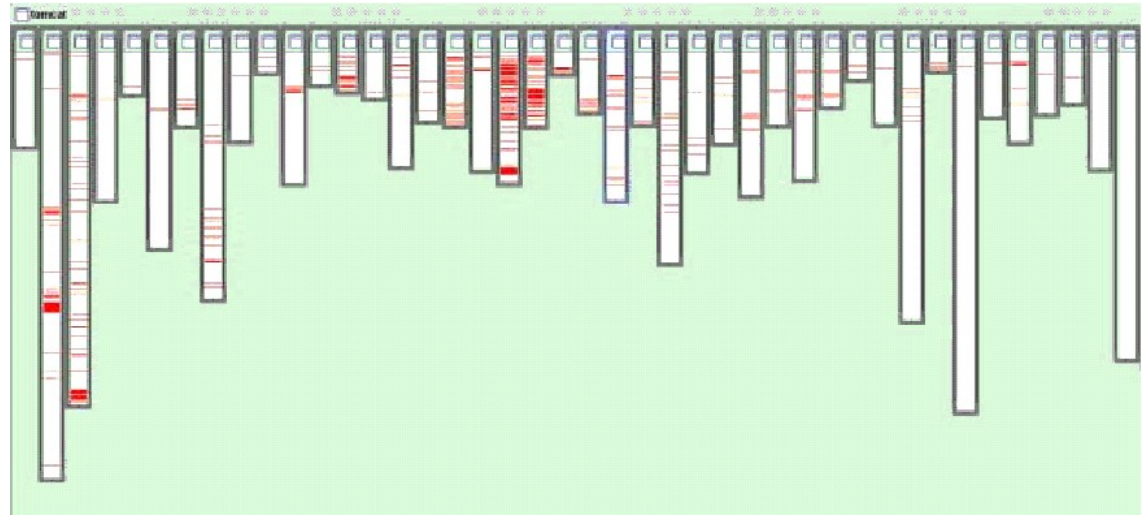
AOP

- Aspect-oriented programming
- „separation of concerns“
 - concern ~ a part of program code related to a particular functionality
- typically understood as an extension of OOP
- solves the problem that it is not always possible to put a code for a single functionality to a single (or several) classes
 - contrary, code is through the all application

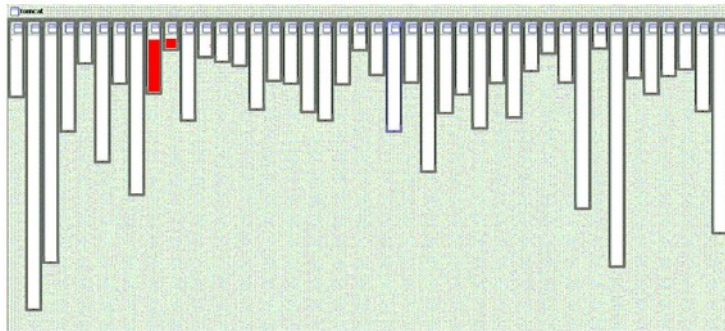
Application modularity



XML parsing in Tomcat



logging



URL manipulating in Tomcat

AspectJ

- <http://www.eclipse.org/aspectj/>
- a Java extension
 - 1 concept – **joinpoint**
 - a place in a program for adding code
 - several constructs
 - **pointcut**
 - definition of joinpoint(s)
 - **advice**
 - code to be added
 - **inter-type declaration**
 - extending a class declaration
 - **aspect**
 - a “class” that can contain the above mentioned constructs

Pointcut

- `call(void Point.setX(int))`
- `call(void Point.setX(int)) ||`
`call(void Point.setY(int))`
- `call(void FigureElement.setXY(int,int)) ||`
`call(void Point.setX(int)) || call(void Point.setY(int)) ||`
`call(void Line.setP1(Point)) ||`
`call(void Line.setP2(Point))`
- `pointcut move():`
`call(void FigureElement.setXY(int,int)) ||`
`call(void Point.setX(int)) || call(void Point.setY(int)) ||`
`call(void Line.setP1(Point)) ||`
`call(void Line.setP2(Point));`
- `call(public * Figure.*(..))`

Advice

- `before(): move() {
 System.out.println("about to move");
}`
- `after() returning: move() {
 System.out.println("just successfully moved");
}`

Inter-type declaration

- aspect PointObserving {
 private Vector Point.observers = new Vector();
 ...
}

Aspect

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    pointcut changes(Point p): target(p) && call(void Point.set*(int));
    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }

    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}
```


Aspect

- aspect SimpleTracing {
 pointcut tracedCall():
 call(void FigureElement.draw(GraphicsContext));

 before(): tracedCall() {
 System.out.println("Entering: " + thisJoinPoint);
 }
}
- aspect SetsInRotateCounting {
 int rotateCount = 0;
 int setCount = 0;

 before(): call(void Line.rotate(double)) {
 rotateCount++;
 }

 before(): call(void Point.set*(int))
 && cflow(call(void Line.rotate(double))) {
 setCount++;
 }
}

AspectJ

- aspects can be defined directly in Java
 - via annotations

```
@Aspect
```

```
public class Foo {
```

```
    @Pointcut("call(* *.*(..))")
```

```
    void anyCall() {}
```

```
    @Before("call(* org.aspectprogrammer.*(..)  
            && this(Foo)")
```

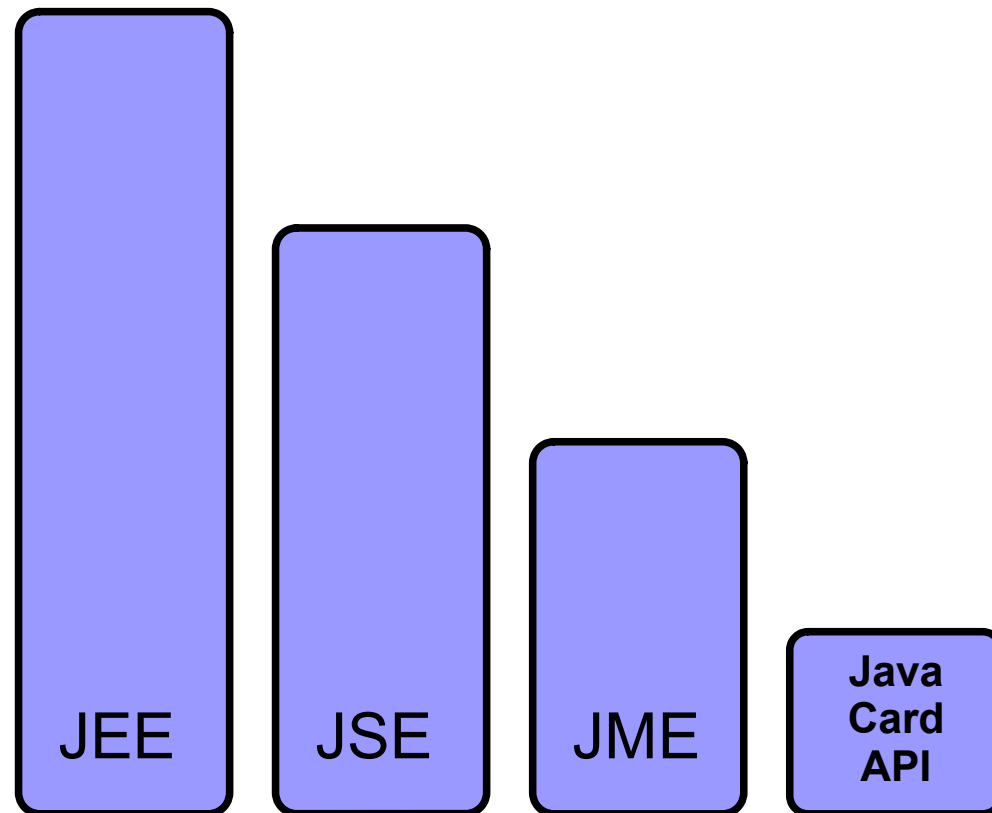
```
        public void callFromFoo() {  
        }
```

```
}
```

JAVA

JEE
Java Enterprise Edition

Overview



“Enterprise” applications

- “big enterprise” applications
- required features
 - re-usability
 - loosely coupled
 - transactions
 - declarative interface
 - persistence
 - security
 - distributed applications
 - ...

3-tier architecture

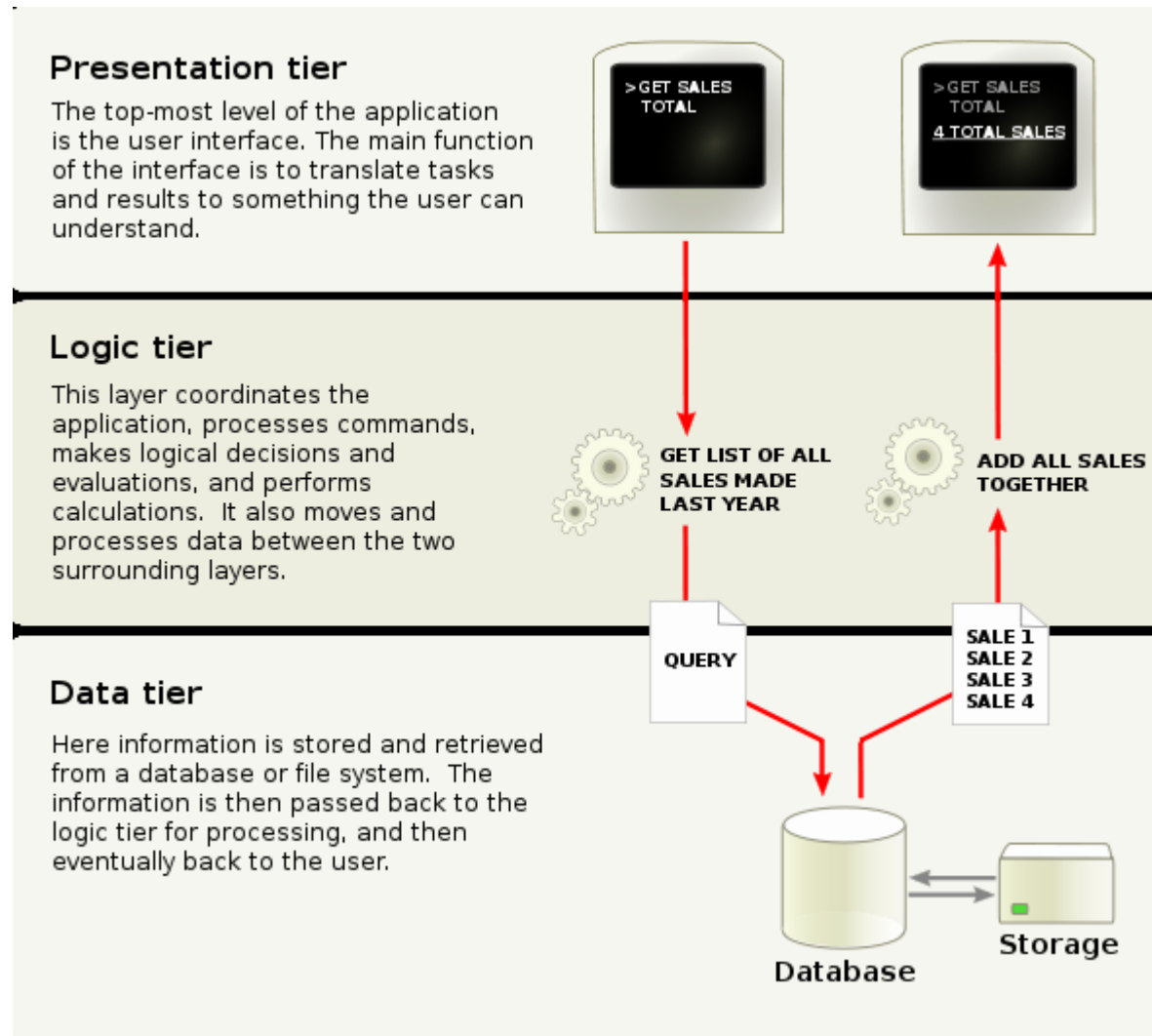


image source: http://en.wikipedia.org/wiki/File:Overview_of_a_three-tier_application_vectorVersion.svg

JAVA

EJB

(first, briefly EJB 2, i.e. old EJB)

Overview

- Enterprise Java Beans
- components
- runs in a server
 - an EJB container
- local and remote access
- the container offers many services
 - persistence
 - security
 - transactions
 - scalability
 - concurrency

EJB

- kinds of beans
 - session beans – implement business logic (logic tier), not persistent
 - stateless – no state kept
 - statefull – a state is kept
 - message-driven beans
 - implements prescribed interface
 - MessageListener – onMessage()
 - entity beans – persistent data
 - persistence
 - container managed
 - bean managed
- deployment descriptor
- EAR

EJB

- many issues
 - necessity to create several interfaces and classes
 - classes had to have the same methods but had not implement the interfaces
 - EJB container “ties” the interface and implementation
 - generates stubs and skeletons
 - necessity to create multiple descriptors
 - ...

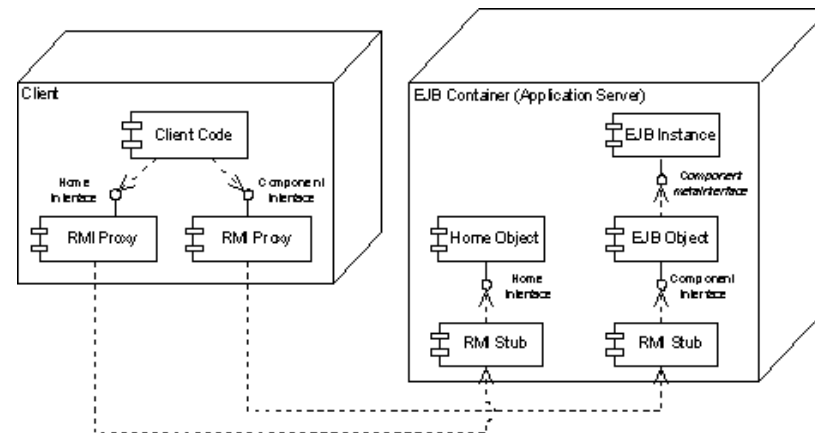


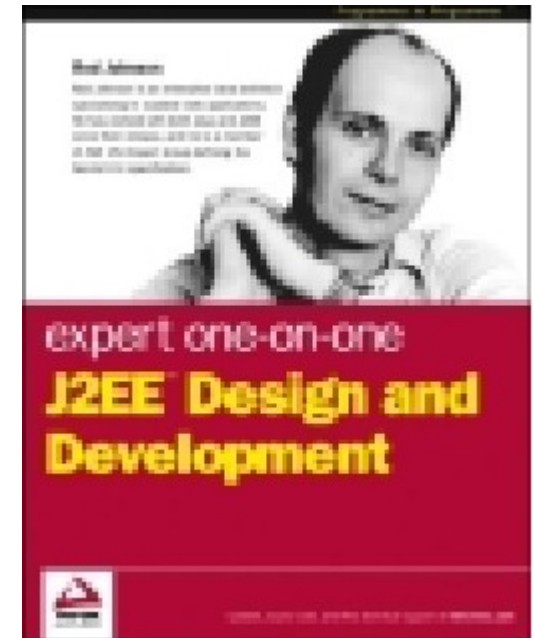
image source: B.Eckel: Thinking in Enterprise Java

JAVA

Spring

Overview

- 2002
- critique of EJB
 - to complex
 - hard to be used
 - hard to be tested
 - RemoteException everywhere
 - ...
- Rod Johnson: Expert One-on-One J2EE Design and Development
 - critique of EJB + proposal of a better architecture
 - Spring foundations



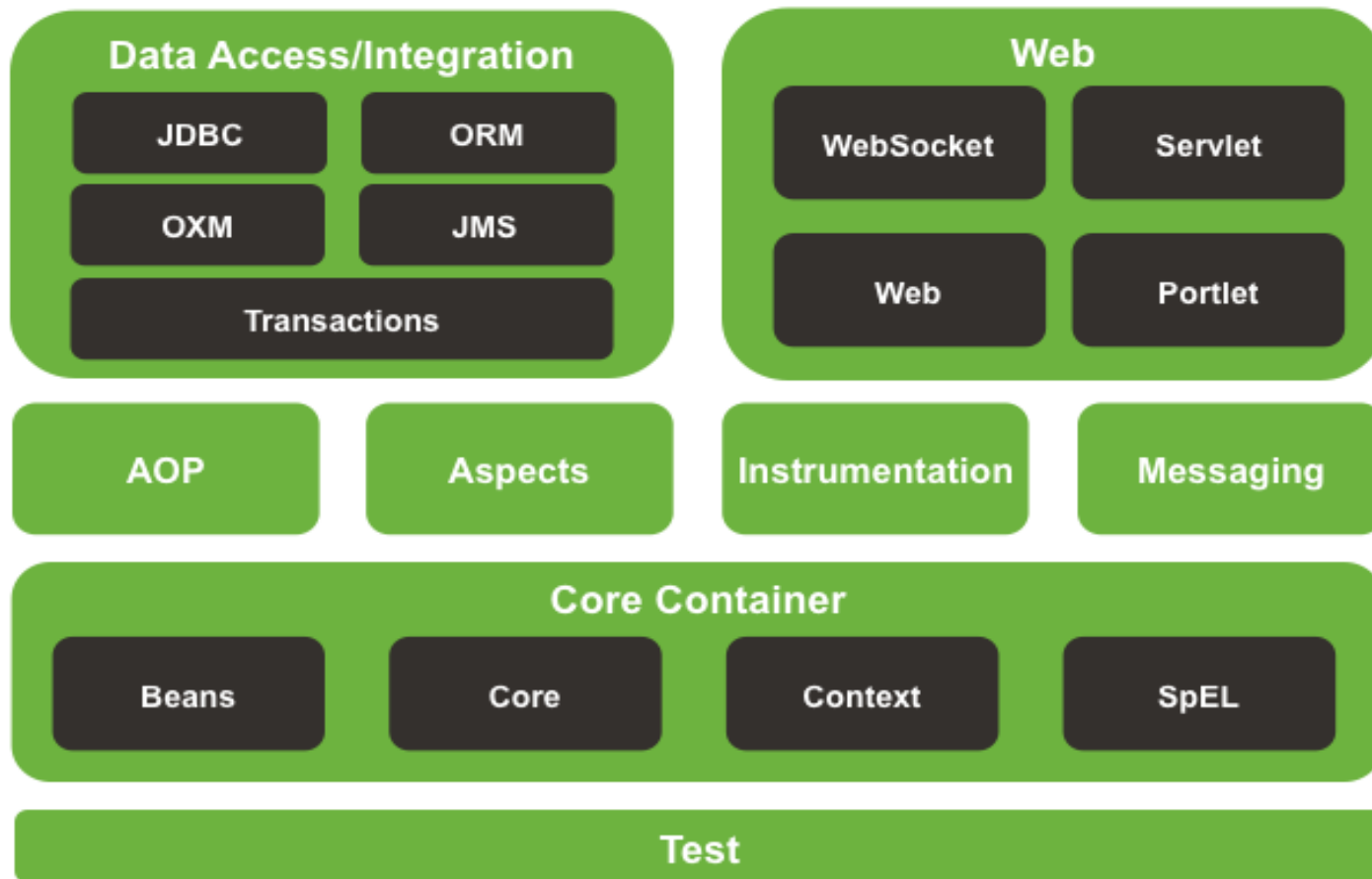
Overview

- Spring
 - <http://www.spring.io/>
 - based on POJO
 - plain old Java objects
 - but can be integrated with EJB
 - “lightweight” solution
 - the smallest possible dependency of application code on Spring
 - no server necessary
 - suitable for any type of application
 - effort for integration with other frameworks
 - not to “reinvent the wheel”
 - to use proven existing solutions

Architecture



Spring Framework Runtime



Spring core

- the `org.springframework.beans` package
- an “inversion of control” container
 - Dependency Injection
 - Hollywood Principle: "Don't call me, I'll call you."
- objects are not interconnected in code but in a configuration file
- an object is not responsible for searching its dependencies
- dependencies are declared
 - a container “provides” them – sets them via setters
 - common naming conventions `setXxx()`
 - or via parameters of constructors
- no special requirements on objects

Spring core

- objects created via a “factory”
 - the interface
`org.springframework.beans.factory.BeanFactory`
 - the most used factories
 - `DefaultListableBeanFactory`

Spring core – example

```
public class nameBean {  
    String name;  
  
    public void setName(String a) {  
        name = a;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

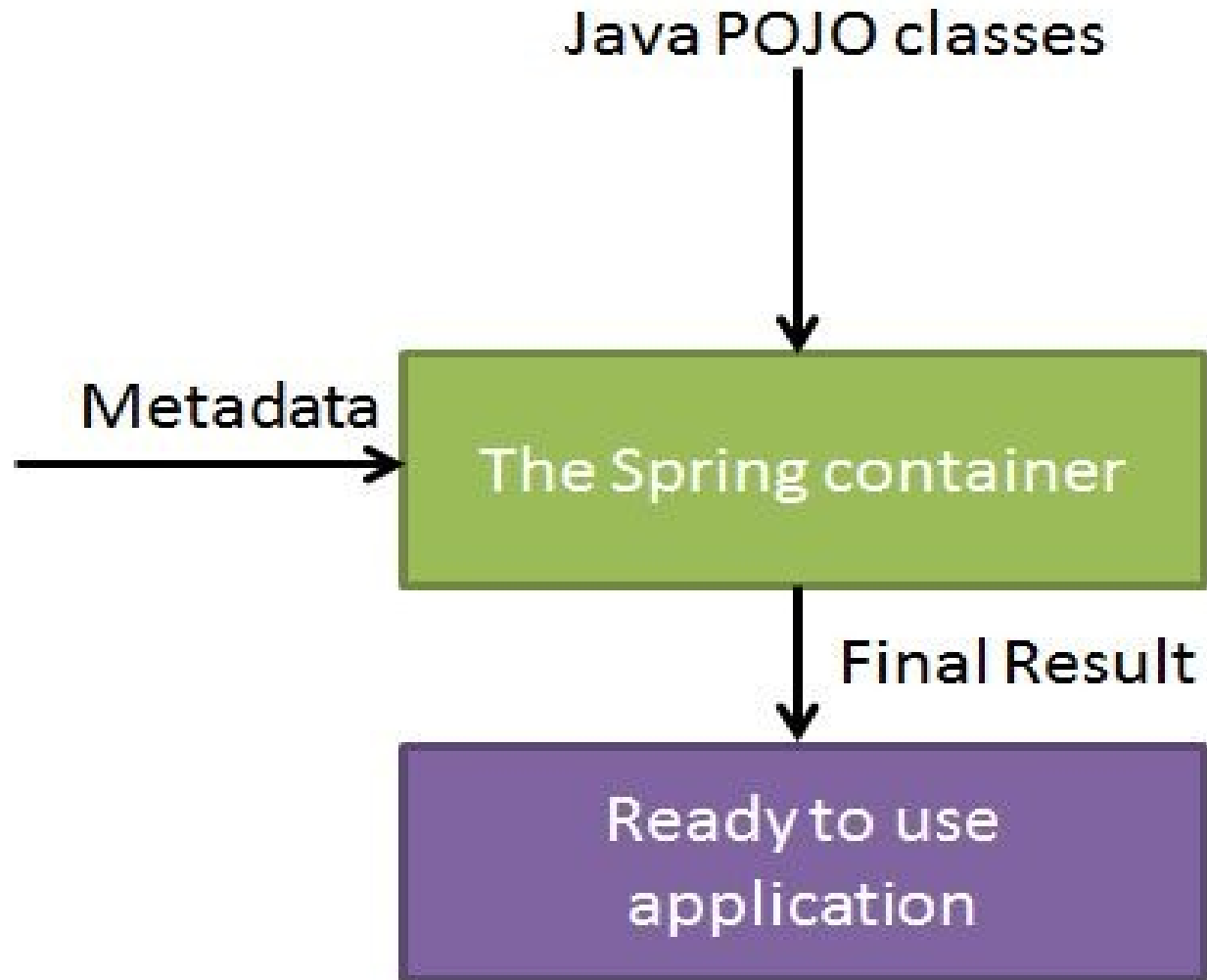
```
<bean id="bean1" class="nameBean">  
    <property name="name" >  
        <value>Tom</value>  
    </property>  
</bean>
```

- interconnecting objects

```
<bean id="bean" class="beanImpl">  
    <property name="conn">  
        <ref bean="bean2"/>  
    </property>  
</bean>
```

```
<bean id="bean2" class="bean2impl"/>
```

Spring core



Spring and data tier

- anything can be used
 - JDBC
 - ORM
 - Hibernate
 - ...
- can be used separately
 - simplified DB usage
 - unified exceptions
 - ...

Spring and data tier

- ```
JdbcTemplate template = new JdbcTemplate(dataSource);
List names = template.query("SELECT USER.NAME FROM USER",
 new RowMapper() {
 public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
 return rs.getString(1);
 }
 });
```
- ```
int youngUserCount = template.queryForInt("SELECT COUNT(0) FROM USER WHERE
USER.AGE < ?", new Object[] { new Integer(25) });
```
- ```
class UserQuery extends MappingSqlQuery {
 public UserQuery(DataSource datasource) {
 super(datasource, "SELECT * FROM PUB_USER_ADDRESS
 WHERE USER_ID = ?");
 declareParameter(new SqlParameter(Types.NUMERIC));
 compile();
 }
 protected Object mapRow(ResultSet rs, int rownum) throws SQLException{
 User user = new User();
 user.setId(rs.getLong("USER_ID")); user.setForename(rs.getString("FORENAME"));
 return user; }
 public User findUser(long id) { return (User) findObject(id); }
}
User user = userQuery.findUser(25);
```

# Spring AOP

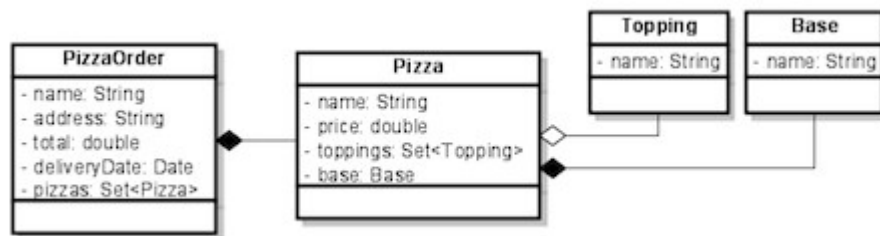
- implemented in plain Java
  - can be integrate with AspectJ
- intended for functionality for which aspects are ideal
  - originally for adding JEE services to Spring
  - transactions
  - logging
  - ...

# Other Spring parts

- Spring MVC
  - a web MVC framework
  - inspired by the Struts framework
  - does not prescribe what should be used for generating pages
    - JSP
    - template systems (Velocity,...)
    - ...
- EJB
  - instead of POJO, EJBs can be used
- ...

# Spring Roo

- framework easy generation of enterprise applications
  - roughly creation of an application using a “wizard” in several steps



The screenshot shows a web browser window with the URL `http://localhost:8080/pizzashop/pizzaorder/form`. The page title is "Welcome to Pizzashop". The interface features a green header with "ROO" and the Spring logo. A sidebar on the left contains navigation links for Topping, Pizza Order, Base, and Pizza. The main content area is titled "Create new Pizza Order" and contains form fields for Name (Stefan Schmidt), Address (My Address), Total (5), and Delivery Date (12/24/2009). A calendar widget is visible for the delivery date. A "SAVE" button is located at the bottom left of the form. The footer includes links for Home, Logout, Language, Theme, and Sponsorship.

# JAVA

## EJB 3



# Overview

- inspired by Spring
- instead of implementing interfaces, annotations are used
- using “dependency injection”
- no need to use descriptors
- ...
- entity beans replaced by Java Persistence API
  - “mapping” classes to tables in relational database
  - JPQL query language
    - “SQL over objects”

# Session bean – example

@Remote

```
public interface Converter {
 public BigDecimal dollarToYen(BigDecimal dollars);
}
```

@Stateless

```
public class ConverterBean implements converter.ejb.Converter {
 private BigDecimal euroRate = new BigDecimal("0.0070");

 public BigDecimal dollarToYen(BigDecimal dollars) {
 BigDecimal result = dollars.multiply(yenRate);
 return result.setScale(2, BigDecimal.ROUND_UP);
 }
}
```

# Message-driven bean – example

```
@MessageDriven(mappedName="MDBQueue")
public class MDB implements MessageListener {
 public void onMessage(Message msg) {
 System.out.println("Got message!");
 }
}
```

# Entity – example

```
@Entity
@Table(name = "phonebook")
public class PhoneBook implements Serializable {
 @Column(name="number") private String number;
 @Column(name="name") private String name;

 public PhoneBook() {}

 public PhoneBook(String name, String number) {
 this.name = name;
 this.number = number;
 }

 @Id public String getName() { return name; }
 public void setName(String name) { this.name = name; }
 public String getNumber() { return number; }
 public void setNumber(String number) { this.number = number; }
}
```

# JPQL

- inspired by HQL
  - a subset of HQL
- `SELECT ... FROM ...`
  - `[WHERE ...]`
  - `[GROUP BY ... [HAVING ...]]`
  - `[ORDER BY ...]`
- `DELETE FROM ... [WHERE ...]`
- `UPDATE ... SET ... [WHERE ...]`
  
- `SELECT a FROM Author a ORDER BY a.firstName, a.lastName`
- `SELECT DISTINCT a FROM Author a INNER JOIN a.books b WHERE b.publisher.name = 'MatfyzPress'`

# JAVA

## Hibernate

# Architecture

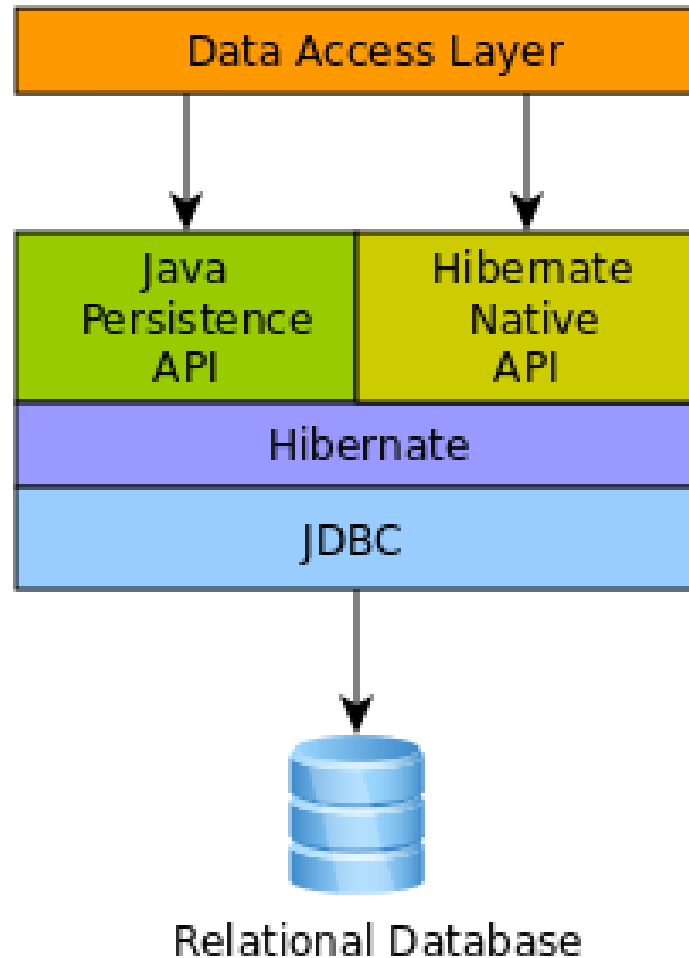


image source: [http://docs.jboss.org/hibernate/orm/5.4/userguide/html\\_single/Hibernate\\_User\\_Guide.html](http://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html)

# Core API

- Session
  - interconnection between DB and application
  - keeps inside a connection to DB
    - a JDBC connection
  - manages objects
    - contains a cache of objects
- SessionFactory
  - a session creator
  - contains mapping between objects and DB
  - can contain a cache of objects
- persistent objects
  - POJOs
  - should follow JavaBeans rules
    - but it is not necessary



# Usage

- roughly
  - creating a configuration
    - XML
  - creating classes
    - Java
  - creating a mapping
    - XML, or
    - Java annotations

# Configuration

- an XML file
- defines
  - a DB connection
  - a type of DB (dialect)
  - a mapping reference
  - ...

```
<hibernate-configuration>
 <session-factory>
 <property name="connection.driver_class">org.h2.Driver</property>
 <property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
 <property name="connection.username">sa</property>
 <property name="connection.password"/>

 <property name="connection.pool_size">1</property>

 <property name="dialect">org.hibernate.dialect.H2Dialect</property>

 <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

 <property name="show_sql">>true</property>

 <property name="hbm2ddl.auto">create</property>

 <mapping resource="org/hibernate/tutorial/hbm/Event.hbm.xml"/>
 </session-factory>
</hibernate-configuration>
```

# Classes for persistent data

- POJO
- should follow JavaBeans rules for naming
  - it is not necessary
- a constructor without parameters is necessary
  - its visibility is not important

```
public class Event {
 private Long id;
 private String title;
 private Date date;

 public Event() {}

 public Event(String title, Date date) {
 this.title = title;
 this.date = date;
 }

 public Long getId() { return id; }
 private void setId(Long id) { this.id = id; }

 public Date getDate() { return date; }
 public void setDate(Date date) { this.date = date; }

 public String getTitle() { return title; }
 public void setTitle(String title) { this.title = title; }
}
```

# Mapping

- an XML file
- mapping class attributes and columns
- defines
  - name
  - type
    - not necessary if it is “obvious”
    - Hibernate types
      - nor Java nor SQL types
      - they are “converters” between Java and SQL types
  - column
    - not necessary if it is the same as the name

```
<hibernate-mapping package="org.hibernate.tutorial.hbm">
 <class name="Event" table="EVENTS">
 <id name="id" column="EVENT_ID">
 <generator class="increment"/>
 </id>
 <property name="date" type="timestamp" column="EVENT_DATE"/>
 <property name="title"/>
 </class>
</hibernate-mapping>
```

# Mapping

```
@Entity
@Table(name = "EVENTS")
public class Event {
 private Long id;
 private String title;
 private Date date;

 public Event() { }

 public Event(String title, Date date) {
 this.title = title;
 this.date = date;
 }

 @Id
 @GeneratedValue(generator="increment")
 @GenericGenerator(name="increment", strategy = "increment")
 public Long getId() { return id; }

 private void setId(Long id) { this.id = id; }

 @Temporal(TemporalType.TIMESTAMP)
 @Column(name = "EVENT_DATE")
 public Date getDate() { return date; }

 public void setDate(Date date) { this.date = date; }

 public String getTitle() { return title; }
 public void setTitle(String title) { this.title = title; }
}
```

- mapping can be defined using annotations
- in the configuration, the class is referenced

# Usage

- ```
SessionFactory sessionFactory =  
    new Configuration().configure().buildSessionFactory();
```
- ```
Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(new Event("Our very first event!", new Date()));
session.save(new Event("A follow up event", new Date()));
session.getTransaction().commit();
session.close();
```
- ```
List result = session.createQuery( "from Event" ).list();
```

States of objects

- Transient
 - created object (new)
 - not yet associated with a Hibernate session
- Persistent
 - the object is associated with a session
 - created and then saved or loaded
- Detached
 - a persistent object but its session was terminated
 - can be associated with a new session

Using objects

- loading
 - `sess.load(Event.class, new Long(id));`
 - an exception is thrown if the object does not exist
 - may not immediately access DB
 - `sess.get(Event.class, new Long(id));`
 - returns null if the object does not exist
- querying
 - `sess.createQuery(...).list()`
- changing objects
 - `Event e = sess.load(Event.class, new Long(69));`
`e.set...`
`sess.flush();`

Using objects

- modifying detached objects
 - `Event e = sess.load(Event.class, new Long(69));`
`e.set...`
 - ...
 - `secondSess.update(e);`
- deleting objects
 - `sess.delete(e);`

Querying

- HQL – Hibernate query language
 - similar to SQL

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

- native SQL can be used too

```
sess.createQuery("SELECT * FROM CATS").list();
```

Hibernate...

- other parts
 - creating classes from tables
 - support for full-text searching
 - object versioning
 - object validation
 - support of JPA (Java Persistence API)
 - ...



Slides version AJ11.en.2020.01

This slides are licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).