Internetanwendungstechnik

Java RMI und Java EE

Gero Mühl

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Kommunikations- und Betriebssysteme (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

Java RMI

- Java Remote Method Invocation (Java RMI)
  - Ermöglicht Methodenaufrufe an Java Objekten, die sich in anderen Java Virtual Machines (JVMs) befinden
- Eigenschaften
  - Bietet Verteilungs- und Lokationstransparenz für Clients
  - Entwickelt für Java → keine extra IDL notwendig
  - Java Security Mechanismen finden Anwendung
  - Dynamisches Laden von Bytecode (Klassen) bei Bedarf (z.B. Stubs)
  - Garbage Collection unreferenzierter Objekte
  - HTTP-Tunneling zur Überwindung von Firewalls möglich

Parameterübergabe

- Remote-Objekte werden per Referenz übergeben
- Instanz der Stub-Klasse wird anstelle des Remote-Objektes übergeben
- Sonstige Objekte werden per Kopie übergeben
- Übergebene Objekte müssen serialisierbar sein → Implementieren des Marker-Interfaces java.io.Serializable
- Bytecode von lokal nicht verfügbaren Klassen kann dynamisch nachgeladen werden

Entwicklung verteilter Anwendungen

1. Schnittstelle definieren und implementieren (Remote-Schnittstelle und Remote-Klasse)
2. Stubs und Skeletons mit rmic kompilieren sofern für Abwärtskompatibilität benötigt
   - Skeletons seit Java 1.2 nicht mehr erforderlich
   - Stubs seit Java 1.5 nicht mehr erforderlich
3. RMI Registry starten
4. Server implementieren, kompilieren und starten
5. Client implementieren, kompilieren und starten

Java RMI Registry

- Speichert Informationen über Remote-Objekte (nicht persistent)
- Start der RMI Registry
  - Von Kommandozeile: `rmiregistry [port]`
  - Durch Server: `LocateRegistry.createRegistry(port);`
- Benutzt URL-basierte Namen, z.B.
  `[rmi://]myhost:port/someJavaobject`
- Server registrieren Remote-Objekte
  `bind(), unbind(), rebind()`
- Clients lösen Namen auf oder suchen Remote-Objekte
  `lookup(), list()`

Java RMI Beispiel – Schnittstelle

```
import java.rmi.*;
import java.rmi.server.*;


// Defining a Remote Interface
public interface Calculator extends Remote {

    double add(double a,double b)
        throws RemoteException;

…

}
```

Marker-Schnittstelle

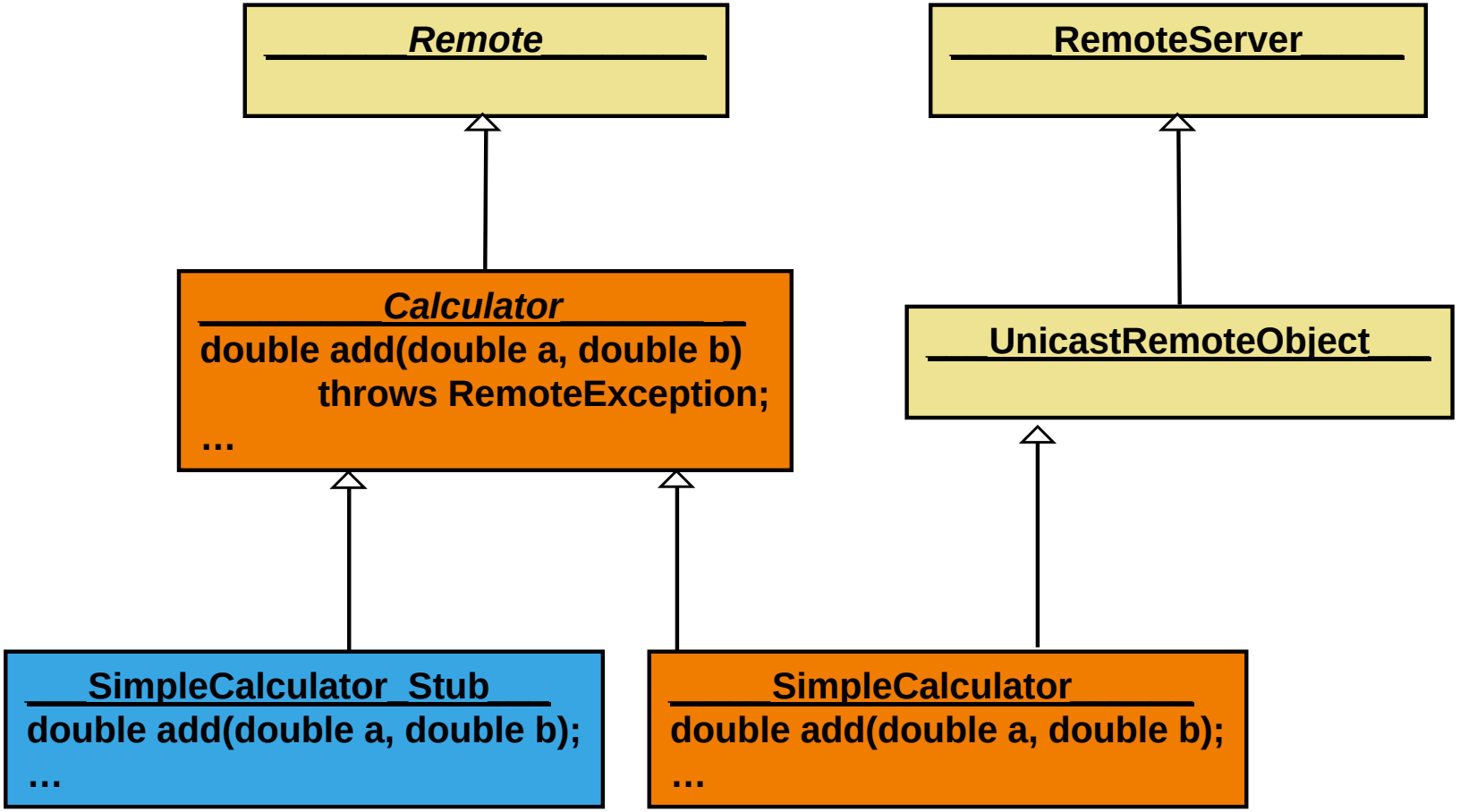
Java RMI Beispiel – Remote-Klasse

```
import java.rmi.*;
import java.rmi.server.*;

// Implementing a Remote Interface
public class SimpleCalculator
    extends UnicastRemoteObject
    implements Calculator {

    public SimpleCalculator
        throws RemoteException {};

    public double add(double a,
                      double b) {
        return a+b;
    }

    …
}
```

UnicastRemoteObject-Klasse

- Bietet Unterstützung zur Erzeugung und zum Export von Remote-Objekten
- Default-Konstruktor exportiert erzeugtes Objekt
- Nur exportierte Objekte können eingehende Aufrufe entgegennehmen
- Objekte können auch manuell exportiert werden

Java RMI Beispiel: Server-Seite



Java RMI Beispiel – Server

```
import java.rmi.*;
import java.rmi.server.*;

public class CalculatorServer {

  public static void main(String[] args) {

    // Set SecurityManager
    System.setSecurityManager(new RMISecurityManager());

    try {

      // Create remote object
      SimpleCalculator calc = new SimpleCalculator();

      // Register remote object with RMI Registry
      Naming.rebind("//myhost/Calculator", calc);

    } catch (Exception e) { … }

    // Process keeps running!

} }
```

Java RMI Beispiel – Client

```
import java.rmi.*;

public class CalculatorClient {

  public static void main(String args[]) {

    // Set SecurityManager
    System.setSecurityManager(new RMISecurityManager());

    try {

      // obtain URL for remote object and
      String name = "//" + args[0] + "/Calculator";
      // lookup remote object (stub is loaded
      // automatically, if not available)
      Calculator calc = (Calculator)Naming.lookup(name);

      // call method on remote object
      double r = calc.add(5,5);

    } catch (Exception e) { … }

} }
```

Dynamisches Nachladen von Bytecode

- Objekte wie Parameter oder Rückgabewerte werden zur Übergabe serialisiert
- RMI annotiert serialisierte Objekte mit ihrer Codebase → `java.rmi.server.codebase`-Eigenschaft
- Codebase enthält eine Liste kommaseparierter URLs
- URLs werden benutzt um den Bytecode der serialisierten Klassen auf Empfängerseite nachzuladen
  - Sofern Bytecode nicht lokal verfügbar ist
  - Notwendig wenn eine Subklasse des deklarierten Parameters oder Rückgabetyps übergeben wird
- Setzen der Codebase-Eigenschaft für eine JVM mit `java –Djava.rmi.server.codebase=http://myhost/mydir …`
- Jedes von der Java Plattform unterstützt URL-Protokoll (z.B. HTTP, FTP, FILE, …) kann genutzt werden

Java RMI Security

- Java Security Manager
  - Kontrolliert Zugang zu sicherheitskritischen Ressourcen wie dem Dateisystem, dem Netzwerk, der GUI, …
  - Kontrolliert Restriktionen für das dynamische Laden von Klassen
- Der Standard Security Manager erlaubt kein dynamisches Nachladen von Java Bytecode
- Notwendigkeit zur Installation eines RMI Security Managers
  - `System.setSecurityManager(new RMISecurityManager())`
  - Setzen entsprechender Sicherheitsrichtlinien

Sicherheitsrichtlinien

- Sicherheitsrichtlinien werden durch eine Policy-Datei gesetzt

```
grant {
    permission java.net.SocketPermission
        "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission
        "*:80", "connect";
};
```

- Werkzeug zum Erstellen einer Policy-Datei → `policytool`
- Setzen der Policy-Datei für eine JVM

```
java –Djava.security.policy=mypolicyfile …
```

Java RMI Security und Stubs

- Problem: Nachladen manipulierter Stub-Klassen
  - Ausgangspunkt für Angriffe
  - Gefährdung der Sicherheit des Clients
- Lösung durch Einschränkung der Rechte → RMI Stubs dürfen
  - Keine (TCP-) Verbindungen (auf Ports) erwarten und annehmen
  - Keine Threads manipulieren, die außerhalb ihrer Thread-Gruppe liegen
  - Keine neuen Prozesse erzeugen
  - Keinen neuen ClassLoader erzeugen
  - Keine dynamische Bibliothek binden
  - Nicht die JVM beenden
  - Nicht auf das Dateisystem zugreifen
  - …

Literatur

1. Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, Revision 1.8, 2002. **http://java.sun.com/products/jdk/rmi/**

2. Sun Microsystems, Inc. *The Java Tutorial*. **http://java.sun.com/docs/books/tutorial/**

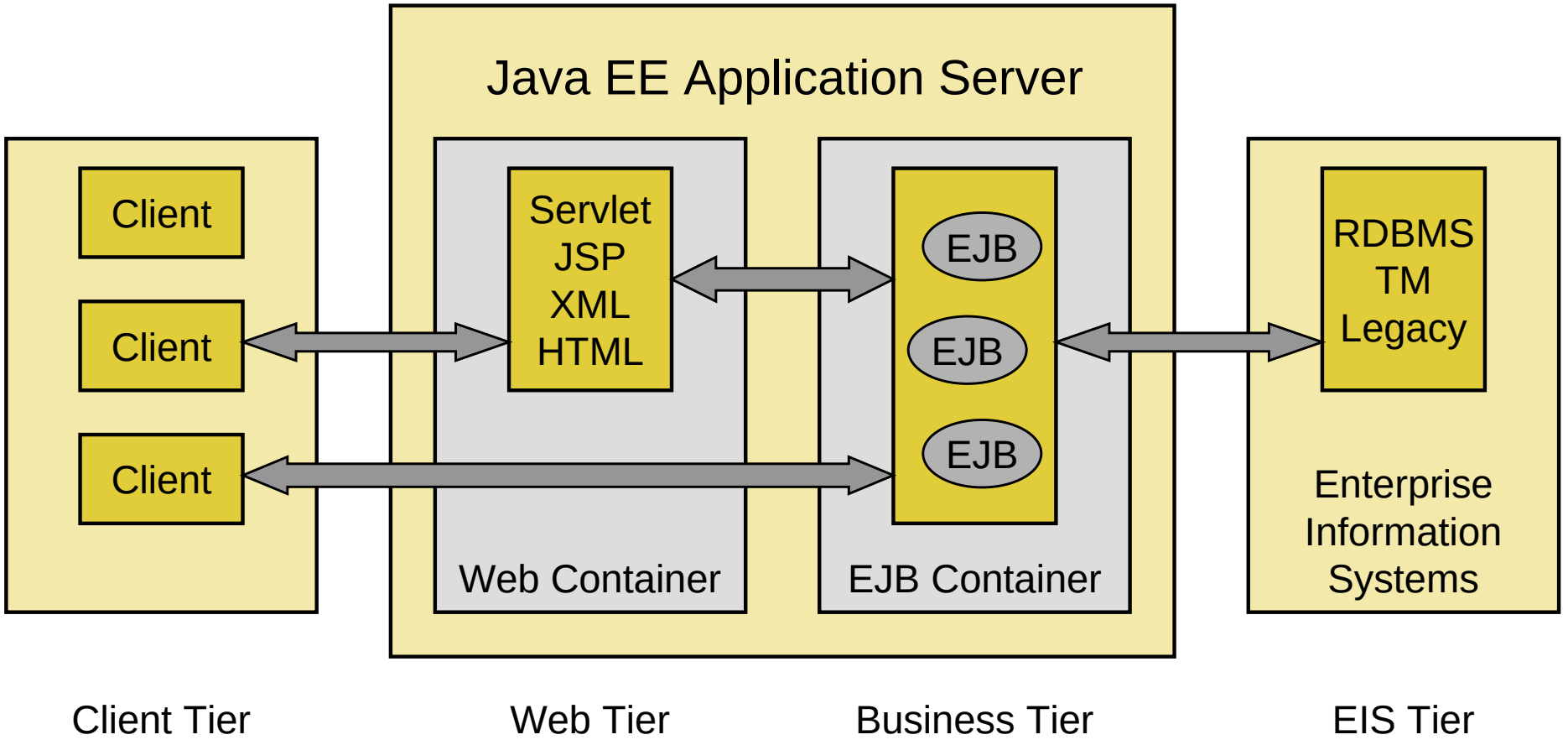
Java Enterprise Edition (Java EE)

- Rahmenwerk zum Design, zur Entwicklung und zum Deployment von mehrschichtigen, verteilten Enterprise-Anwendungen

- Java EE Anwendungsschichten
  - Client-Tier → auf dem Client-Rechner
  - Web-Tier → Java EE Application Server
  - Business-Tier → Java EE Application Server
  - Enterprise Information System (EIS)-Tier → Datenbank-Server

- Java EE umfasst mehrere Technologien
  - Web-Clients, Client-Anwendungen und Applets → Client Tier
  - Servlets und Java Server Pages (JSPs) → Web Tier
  - Enterprise Java Beans (EJBs)-Komponenten → Business Tier

Java EE Architektur

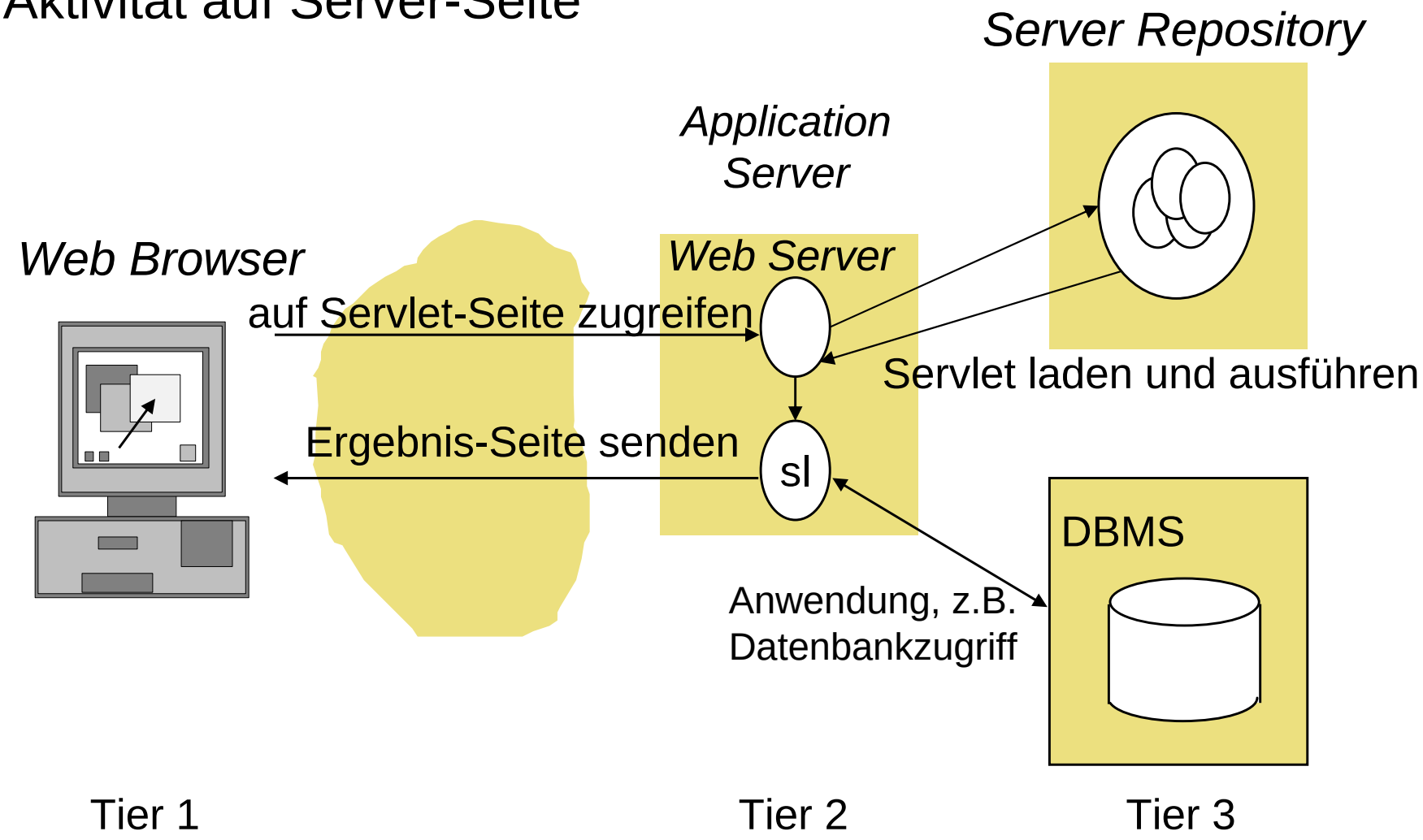


Java Servlets

- Servlets = CGI auf „Java-Art“ [http://java.sun.com/products/servlet/]
- Zustandslose Java-Klassen, die über einen Web Server mittels einer URL angesprochen werden
- Ausführungsschritte
  - Analyse der Request-Parameter
  - Bearbeiten des Request
  - Erzeugen der Ausgabe (Response)
- Benutzen Container-Dienste
  - Session-Management
  - Authentifizierung und Autorisierung
  - …
- Java Packages
  - javax.Servlet
  - javax.HttpServlet

Servlets

- Aktivität auf Server-Seite



HttpServlets

- Implementieren `javax.servlet.http.HttpServlet`
  - Abgeleitet von `GenericServlet`, welches `Servlet` implementiert
- Überschreiben von Methoden für die HTTP-Operationen
  - `doGet()`, `doPost()`, `doDelete()`, `doOptions()`, `doPut()`, ...
  - HTTP-Request und -Response als Parameter
- Lifecycle Methoden (geerbt von `GenericServlet`)
  - `init()` → Initialisierung des Servlets
  - `destroy()` → Freigeben benutzter Ressourcen

javax.servlet.http.HttpServletRequest

- Repräsentiert die HTTP-Anfrage vom Client
- Ist eine Java Bean
- Properties (Getter/Setter Methoden)
  - `Enumeration getAttributeNames()`,
    `Object getAttribute(String name)`,
    `void setAttribute(String name, Object o)`
  - method, header, headerNames, servletPath, requestURI, contentType, contentLength, cookies, …
  - Alternativ als Stream: `reader`
- Dient der Kommunikation zwischen Servlets/JSP
  - Zusätzliche Attribute einfügen/existierende löschen

javax.servlet.http.HttpServletResponse

- Repräsentiert die HTTP-Antwort an den Client
- Ist eine Java Bean mit vielen Properties
  → vgl. HttpServletRequest
- Inhalte
  - `setContentType(String type)`, `setContentLength(int len)`
  - Ausgabestrom: `getWriter()`
- Statuscode, Fehler, Umlenkung
  - `setStatus(SC_NO_CONTENT)`
  - `sendRedirect("http://kbs.cs.tu-berlin.de")`

Servlet Beispiel 1

```
public class ServletA extends HttpServlet {

  public void doGet(HttpServletRequest req,
                    HttpServletResponse res)
              throws ServletException, IOException {

    PrintWriter out = res.getWriter();
    res.setContentType("text/html");

    out.println("<html><head>");
    out.println("<title>Servlet Beispiel</title>");
    out.println("</head><body>");
    out.println("Datum und Zeit:");
    out.println(new java.Util.Date());
    out.println("</body></html>");

  }
}
```

Servlet Beispiel 2

```
<FORM METHOD="POST" ACTION="/servlet/FormServlet">
  <INPUT NAME="firstname" TYPE="text" SIZE="30">
  <INPUT NAME="lastname"  TYPE="text" SIZE="30">
  <INPUT NAME="age"       TYPE="text" SIZE="3">
  <INPUT TYPE="SUBMIT">
</FORM>
```

HTML-Formular

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
              throws ServletException, IOException {
  firstname = request.getParameter("firstname");
  lastname  = request.getParameter("lastname");
  ageString = request.getParameter("age");
  …
}
```

Servlet

Servlets Deploymentdescriptor (web.xml)

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems,
    Inc.//DTD Web Application 2.2//EN„
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>

  <servlet>
    <!-- Servlet Konfiguration … -->
    <servlet-name>TestServlet</servlet-name>
    <servlet-class>kbs.TestServlet</servlet-class>
  </servlet>

  <!-- weitere Servlets/JSP -->
  ...

</web-app>
```

XML Datei

Servlets Assembly

- Web Archive (WAR)
  - WAR bildet einen Kontext/Application
  - WAR-Datei ist im Prinzip eine JAR-Datei

- Struktur

```
index.html
META-INF/
    MANIFEST.MF
WEB-INF/
    web.xml
    lib/
    classes/
        kbs/
            TestServlet.class
```

Servlets Session Management

- Problem
  - HTTP und Servlets sind zustandslos
  - Viele Anwendungen benötigen aber Zustand (z.B. per-Client)
    - Personalisierte Aktienkursliste
    - Einkaufswagen
    - Authentifizierung
    - …
- Lösung
  - Servlet Container bietet Session Management als Service an

Servlets Sessions

- Repräsentieren Client/Server-Beziehung
- Werden auf Cookies oder URL abgebildet
- Klasse `javax.servlet.http.HttpSession`
  - `HttpServletRequest.getSession()`
  - `HttpServletRequest.getSession(boolean create)`
- Anwendungsdaten als benannte Attribute an Sessions gebunden
  - `Session.setAttribute(name,value)`
  - `value = session.getAttribute(name)`
  - Werte müssen serialisierbar sein

Servlets Sessions

- Sessions „fressen“ Betriebsmittel (Speicher)
- Explizite Deaktivierung
  - `HttpSession.invalidate()`
- Globaler Timeout in Servlet Runtime
- Per-Session Timeout
  - `HttpSession.setMaxInactiveTimeout()`

Servlets Ereignisse

- Ereignisse (Events) ermöglichen über Änderungen fortlaufend unterrichtet zu werden
- Tritt ein Ereignis auf, so wird der Listener aufgerufen
- Registrierung eines Listeners erfolgt im Deployment-Deskriptor

```
<listener>

  <listener-class>
    myApp.MySessionAttributeListenerClass
  </listener-class>

</listener>
```

Servlets Ereignisse

- `HttpSessionBindingListener` Interface

```
void valueBound (HttpSessionBindingEvent event)
void valueUnbound (HttpSessionBindingEvent event)
```

  - Implementiert von Attribut-Objekten
  - Event wenn Objekt gebunden, entfernt oder `session.invalidate()`

- `HttpSessionAttributeListener` Interface

```
void attributeAdded (HttpSessionBindingEvent se)
void attributeRemoved (HttpSessionBindingEvent se)
void attributeReplaced (HttpSessionBindingEvent se)
```

- `HttpSessionBindingEvent` Klasse

```
String getName()
HttpSession getSession()
Object getValue()
```

Servlets Ereignisse

- `HttpSessionActivationListener` Interface

  ```
  void sessionDidActivate(HttpSessionEvent se)
  void sessionWillPassivate(HttpSessionEvent se)
  ```

  - Implementiert von Attribut-Objekten
  - Event wenn Session aktiviert oder deaktiviert `HttpSessionEvent`

- `HttpSessionListener` Interface

  ```
  void sessionCreated(HttpSessionEvent se)
  void sessionDestroyed(HttpSessionEvent se)
  ```

- `HttpSessionEvent` Klasse

  ```
  HttpSession getSession()
  ```

Java Server Pages (JSP)

- Erweiterung der Servlet Technologie
- Gemischte Seitenbeschreibung: statische (Static Template Code) und dynamische Inhalte (JSP Elements)
- JSP umfasst Direktiven, Scripting Code und Aktionen
- Eine JSP ist eine andere Notation eines Servlets
- JSP wird in Servlet Klasse kompiliert

JSP Beispiel – HTML + Java Code

```
<html>

  <head>
    <title>JSP Beispiel</title>
  </head>

  <body>

    Datum und Zeit:<%= new java.util.Date()%><br/>

    <% for (int i=0; i<5; i++) { %>
      Das Quadrat von <%= i %> ist <%= i*i %><br/>
    <% } %>

  </body>

</html>
```

JSP Direktiven

- Kommunikation mit der JSP Engine

  `<%@ directive {attribute="value"}* %>`

- JSP 1.0

  `<%@ page import="java.util.*" %>`

  - Importieren von Bibliotheken

  `<%@ page language="java" %>`

  - Einstellungen, die für die Seite gelten.

  `<%@ include file="companyBanner.html" %>`

  - Einbindung statischer Dokumente

  `<%@ taglib uri="/taglib" prefix="mytaglib" %>`

  - Erweiterung der vorhandenen Actions

JSP Direktiven

- language Scriptsprache (java)
- extends Basisklasse des Servlets
- import Liste der zu importierenden Pakete
- session *true*|*false*, benutzt oder erzeugt HttpSession
- errorPage URL der Fehlerseite (bei Exception)
- isErrorPage true|false, Seite ist Fehlerseite
- …

JSP Scripting

- Deklarationen: `<%! declarations %>`
  - Definieren und überschreiben von Methoden, z.B. `jspInit()` and `jspDestroy()`

```
<%! public void jspInit()    { ... } %>
<%! public void jspDestroy() { ... } %>
```

- Scriptlets: `<% java code fragment %>`
  - Wird in den Rumpf der service-Methode eingefügt

```
<% java.util.Date date = new java.util.Date();
   out.println(date); %>
```

- Ausdrücke: `<%= expression %>`
  - Ausdruck auswerten und Ergebnis in String umwandeln
  - String in Seite einfügen

```
<%= date %>
```

JSP Implizite Objekte

Name	Typ
request	`HttpServletRequest`
response	`HttpServletResponse`
session	`HttpSession`
out	`JspWriter`
application	`ServletContext`
config	`ServletConfig`
page	äquivalent zu `this`
exception	`java.lang.Throwable`

- Innerhalb von Scriptlets und Ausdrücken kann auf diese Objekte zugegriffen werden, z.B.
  `<% out.println(request.getRemoteHost()); %>`

JSP Sessions

```
<HTML><BODY>
  <FORM METHOD=POST ACTION="Save.jsp">
    What's your name?
    <INPUT TYPE="TEXT" NAME="username" SIZE="20">
    <INPUT TYPE="TEXT" NAME="email" SIZE="20">
    <INPUT TYPE="SUBMIT">
  </FORM>
</BODY></HTML>
```

Form.html

```
<%
  String name = request.getParameter( "username" );
  session.setAttribute( "username", name );
  String email = request.getParameter( "email" );
  session.setAttribute( "email", email );
%>
```

Save.jsp

JSP <useBean>-Tag

```
public class UserData {
  public void setUsername(String value){username = value;}
  public void setEmail(String value) {email = value;}
  public String getUsername() {return username;}
  public String getEmail() {return email;}
}
```

Bean

```
<jsp:useBean id="user" class="UserData" scope="session"/>
  <jsp:setProperty name="user" property="*"/>
</jsp:useBean>
```

Save.jsp

```
<HTML><BODY>
<jsp:useBean id="user" class="UserData" scope="session"/>
  Name: <%= user.getUsername() %><BR>
  Email: <%= user.getEmail() %><BR>
</jsp:useBean>
</BODY></HTML>
```

Load.jsp

Enterprise Java Beans (EJBs)

- Server-side component model for multi-tier applications
- Not to be confused with "Java Beans"
  - Client-side component model for desktop GUI
- EJBs run in a container which is a a runtime environment within the Application Server
- Container provides transparent support for
  - Lifecycle Management
  - Naming
  - Persistence Management
  - Transactions
  - Security Mechanisms
- Bean developer can concentrate on solving business problems
- Greatly “simplifies” the development of large, distributed applications

Three different Types of EJBs

- Entity beans
  - Model persistent business objects
- Session beans
  - Model business processes without persistent state
- Message-driven beans
  - Execute on receipt of asynchronous messages

Entity Beans

- Model persistent business objects
  (such as a customer, products, orders)
- Data-oriented objects that are stored persistently in a database as a data record
- Long-living: survive a container crash
- Uniquely identified by a primary key (e.g. customer number)
  - Enables the client to locate a particular entity bean
- Lifecycle independent of that of clients
- Shared access by many clients → transactions
- Can have relations with other entity beans

Managing Persistence of Entity Beans

- Container-Managed Persistence (CMP)
  - Container is responsible for storing the persistent state of the bean in the data base
  - Container generates the necessary database calls
  - Bean developer declared persistent fields in deployment descriptor (other fields are considered being transient)
- Bean-Managed Persistence (BMP)
  - Bean itself is responsible for making its state persistent
  - Container does not have to generate data base calls
  - More complicated for programmer but more flexibility

Session Beans

- Model business processes
- Are similar to an interactive session
- Represent an interaction with a client
- Are not stored persistently
- Lifecycle controlled by the EJB container
- Can either be
  - stateful (e.g. shopping cart) or
  - stateless (e.g. calculate a discount)

  across multiple invocations.

Stateless Session Beans

- Do not maintain a conversational state for the client
  - No state that persists across method invocations
- Except during method invocation, all instances of a stateless bean are equivalent
  - EJB container can assign an instance to any client
  - One bean can manage several clients
- Usually provide better performance and scalability than stateful session beans
  - Fewer beans can support the same number of clients
  - Stateless session beans are never written to secondary storage

Stateful Session Beans

- Maintain a conversational state for one client
- Unique instance for each client
- State retained for the duration of the interaction
- When client removes the bean or terminates, the session ends and the state disappears

Message-Driven Beans (since EJB 2.0)

- Message-Driven Beans (MDBs) enable messaging in EJB
- The only EJBs that can receive messages asynchronously
- Designed for using Java Messaging Service (JMS)
- But can also support other messaging middleware
- Execute on the receipt of a single message
  - onMessage() method is called
- Cannot be invoked because they have no remote or home interface

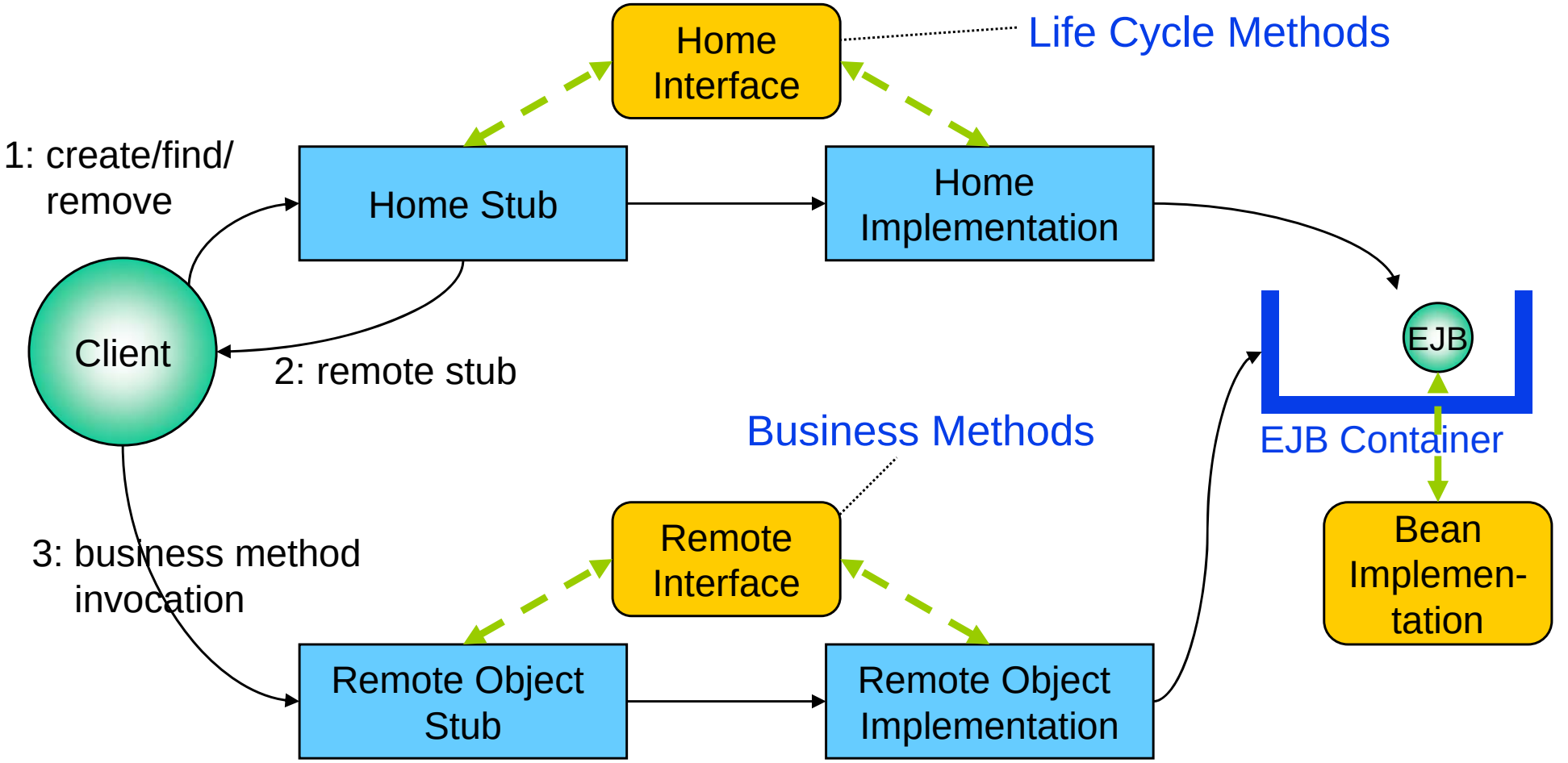
Message-Driven Beans

- Similar to stateless session beans wrt. several aspects
  - they are stateless (but can call other session or entity beans)
  - all instances of a MDB class are equivalent
  - a message is delivered to any instance of a MDB class and processed concurrently
  - a single MDB can process messages from different sources
- Transactions, pooling, and message acknowledgement usually handled by the container
- XA transaction support
  - XA = standard for interface to transactional resources

Bean Home and Remote Interface

- Clients access entity and session beans through interfaces
- For remote clients each bean (except MDBs) has two interfaces
  - Home Interface defines generic lifecycle methods
    - `create(…)`
    - `remove(…)`
    - For entity beans only
      - Finder methods such as `findByPrimaryKey(…)`
      - Home methods that are applied to all instances of a bean.
  - Remote Interface defines the business methods of the bean
- Beans do not directly implement Home or Remote Interfaces
  - Ensures that clients communicate with the bean via the container
- Additional interfaces for local clients (running in the same container)

EJB Home/Remote Interfaces



Developing a Bean

- To deploy a bean, you must provide files containing the following information
  - Deployment descriptor
    - An XML file that specifies information about the bean such as its persistence type and transaction attributes
  - Remote and home interfaces
    - Required for remote access
  - Enterprise bean class
    - A Java class file that implements the methods defined in the interfaces

Example Entity Bean – Remote Interface

```
import java.rmi.*;
import javax.ejb.*;

public interface CustomerRemote extends EJBObject {

  // Defines the business methods that
  // can be invoked by clients

  public String getName() throws RemoteException;
  public void setName(String name) throws RemoteException;

  public String getCity() throws RemoteException;
  public void setCity(String city) throws RemoteException;

  …

} // CustomerRemote
```

Example Entity Bean – Home Interface

```
import java.rmi.*;
import javax.ejb.*;

public interface CustomerHome extends EJBHome {

  // Defines the lifecycle methods

  public Customer create(Integer id)
                  throws CreateException, RemoteException;

  public Customer findByPrimaryKey(Integer id)
                  throws RemoteException, FinderException;

} // CustomerHome
```

Entity Bean Class

- An Entity Bean Class must
  - meet the container-managed persistence syntax requirements
  - be defined as `public` and `abstract`
  - implement the `EntityBean` interface
  - implement zero or more `ejbCreate` and `ejbPostCreate` methods
  - implement the get/set access methods, defined as `abstract`, for the persistent and relationship fields
  - implement any select methods, defining them as `abstract`
  - implement the home methods (home interface)
  - implement the business methods (remote interface)
  - **not** implement the finder methods and the `finalize` method

Example Entity Bean – Bean Class

```
import javax.ejb.EntityBean;

public abstract class CustomerBean implements EntityBean {

  // setter/getter methods for persistent fields
  public abstract Integer getId();
  public abstract void setId(Integer id);
  public abstract String getName();
  public abstract void setName(String name);
  public abstract String getCity();
  public abstract setCity(String city);



  // required for deployment
  public Integer ejbCreate(Integer id) {
    this.setId(id); return null;  // CMP
  }
  public void ejbPostCreate(Integer id){};
```

For each **create** method in the Home Interface there is a `ejbCreate` and `ejbPostCreate` method with the same parameters.

Example Entity Bean – Bean Class

```
  // other lifecycle methods
  public void ejbActivate() {}
  public void ejbLoad() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}
  public void ejbStore() {}

  public void setEntityContext(EntityContext ctx) {}
  public void unsetEntityContext() {}

} // CustomerBean
```

Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC
   "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
  "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Customer</ejb-name>
      <home>CustomerHome</home>
      <remote>CustomerRemote</remote>
      <ejb-class>CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <abstract-schema-name>Customer</abstract-schema-name>
      <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>city</field-name></cmp-field>
```

Persistent Fields

Deployment Descriptor

```
      <primkey-field>id</primkey-field>
      <transaction-type>Container</transaction-type>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Customer</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Container-Managed Persistence

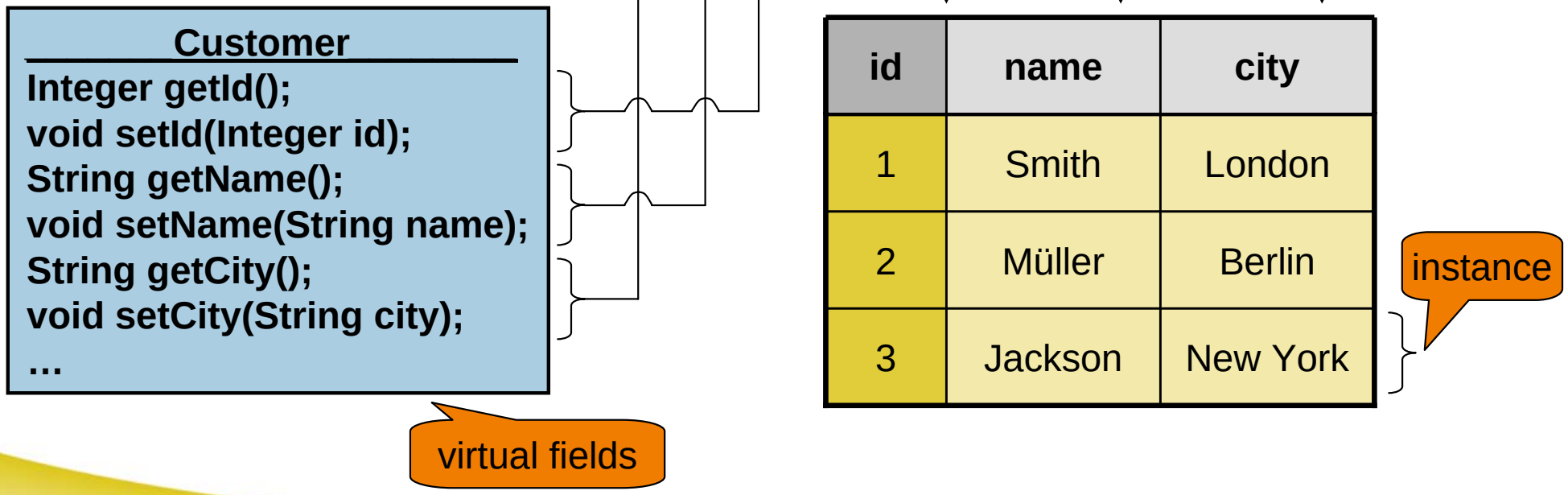
- Bean declares
  - persistent fields using abstract access (getter/setter) methods
  - type of fields (either serializable or primitive)
- A Relational Data Base Management System (RDBMS) is usually used to store the data records
- For each entity bean instance there is a separate record
- Data record is
  - created by calling `create()` or `ejbCreate()`
  - removed by calling `remove()` or `ejbRemove()`
- Container
  - retrieves data record from DB when access is required
  - updates data record in DB when data has changed

Container-Managed Persistence

```
<abstract-schema-name>Customer</abstract-schema-name>
<cmp-field><field-name>id</field-name></cmp-field>
<cmp-field><field-name>name</field-name></cmp-field>
<cmp-field><field-name>city</field-name></cmp-field>
<primkey-field>id</primkey-field>
<transaction-type>Container</transaction-type>
```

persistence schema in deployment descriptor

customer table in DB



Finder

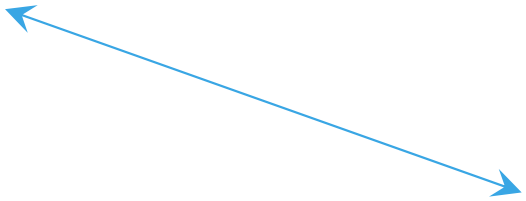
- Search for existing Bean instances
- Are declared in the Home Interface
- Container defines standard-Finder for primary key and all other persistent fields
  `findyBy<FieldName>(<FieldType> s)`
  - e.g. `findByName(String name)`
- Custom-Finders can be defined in the deployment descriptor using *EJB-QL* (SQL-like Query Language)
- Multi-entity finder: `java.util.Collection`

Custom Finder – Bean Class

```
import javax.ejb.*;
import java.util.Collection;

public interface CustomerHome extends EJBHome {

  …

  public Collection findByMaxPrice(double maxPrice)
                    throws RemoteException, FinderException;

  …

}
```

Custom Finder – Deployment Descriptor

```
<query>

  <query-method>
    <method-name>findByMaxPrice</method-name>
    <method-params>
        <method-param>double</method-param>
    </method-params>
  </query-method>

  <ejb-ql>
    SELECT OBJECT(p) FROM Product p WHERE p.price <= ?1
  </ejb-ql>

</query>
```



Example Stateless Session Bean – Remote Interface

```
import javax.ejb.*;
import javax.rmi.*;

public interface PaymentRemote extends EJBObject {

  public void pay(String ccNum, String name,
                  String ccComp, int validYr,
                  int validMnth, double amount)
              throws RemoteException, PaymentFailed;

  …

}
```

Example Stateless Session Bean – Home Interface

```
import javax.ejb.*;
import javax.rmi.*;

public interface PaymentHome extends EJBHome {

  public Payment create()
                  throws CreateException, RemoteException;

  …

}
```

Session Bean Class

- A Session Bean Class must
  - implement the `SessionBean` interface
  - be `public`
  - must not be defined as `abstract` or `final`
  - implement one or more `ejbCreate` methods
  - implement the business methods
  - contain a public no-args constructor
  - **not** define the `finalize` method

Example Stateless Session Bean Class

```
import javax.ejb.*;

public class PaymentBean implements SessionBean {
  public void pay(String ccNum, String name, String ccComp,
                  int validYr, int validMnth, double amount)
               throws PaymentFailed {

    // check arguments

    if (paymentPossible) {

      // add record to DB and commit payment

    } else {

      // log attempt to defraut
      throw new PaymentFailed(PaymentFailed.REP_STOLEN);

} } }
```

Message-Driven Bean Class

- A MDB Class must
  - implement the `MessageDrivenBean` and `MessageListener` interfaces
  - be defined as `public`
  - not be defined as `abstract` or `final`
  - implement one `onMessage` method
  - implement one `ejbCreate` method and one `ejbRemove` method
  - contain a public no-args constructor
  - **not** define the `finalize` method

Example Message-Driven Bean Class

```
import javax.ejb.*;
import javax.jms.*;

public class SimpleMessageBean
             implements MessageDrivenBean, MessageListener {

  private transient MessageDrivenContext mdc = null;
  private Context context;

  public SimpleMessageBean() {} // required for deployment

  public void setMessageDrivenContext(
                                  MessageDrivenContext mdc) {
    this.mdc = mdc;
  }
```

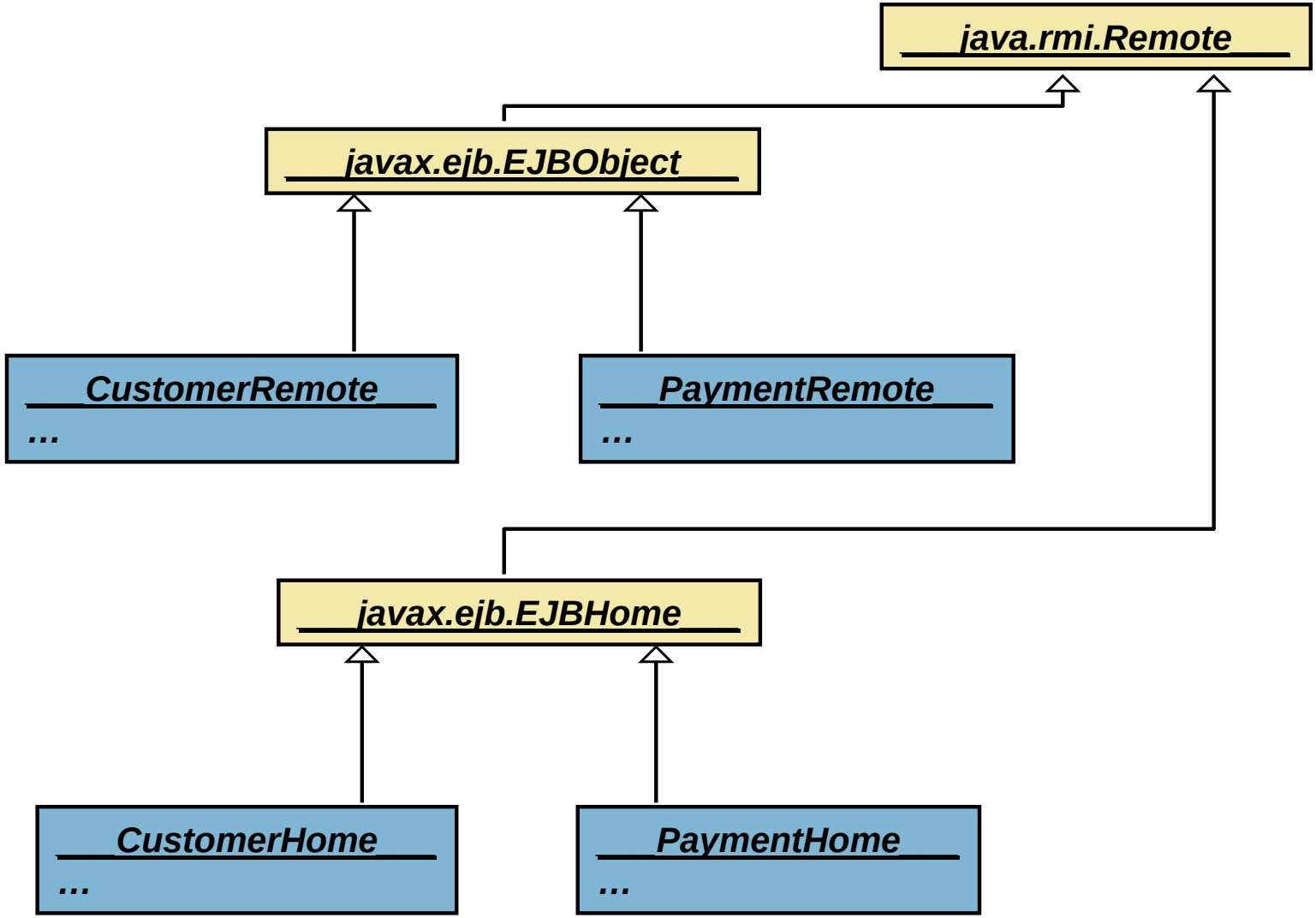
Example Message-Driven Bean Class

```
  public void ejbCreate() {} // required for deployment
  public void ejbRemove() {}

  public void onMessage(Message inMessage) {
    try {
      TextMessage msg = (TextMessage) inMessage;
      logger.info(msg.getText());
    } catch (Exception e) {
      …
  } }

} // class
```

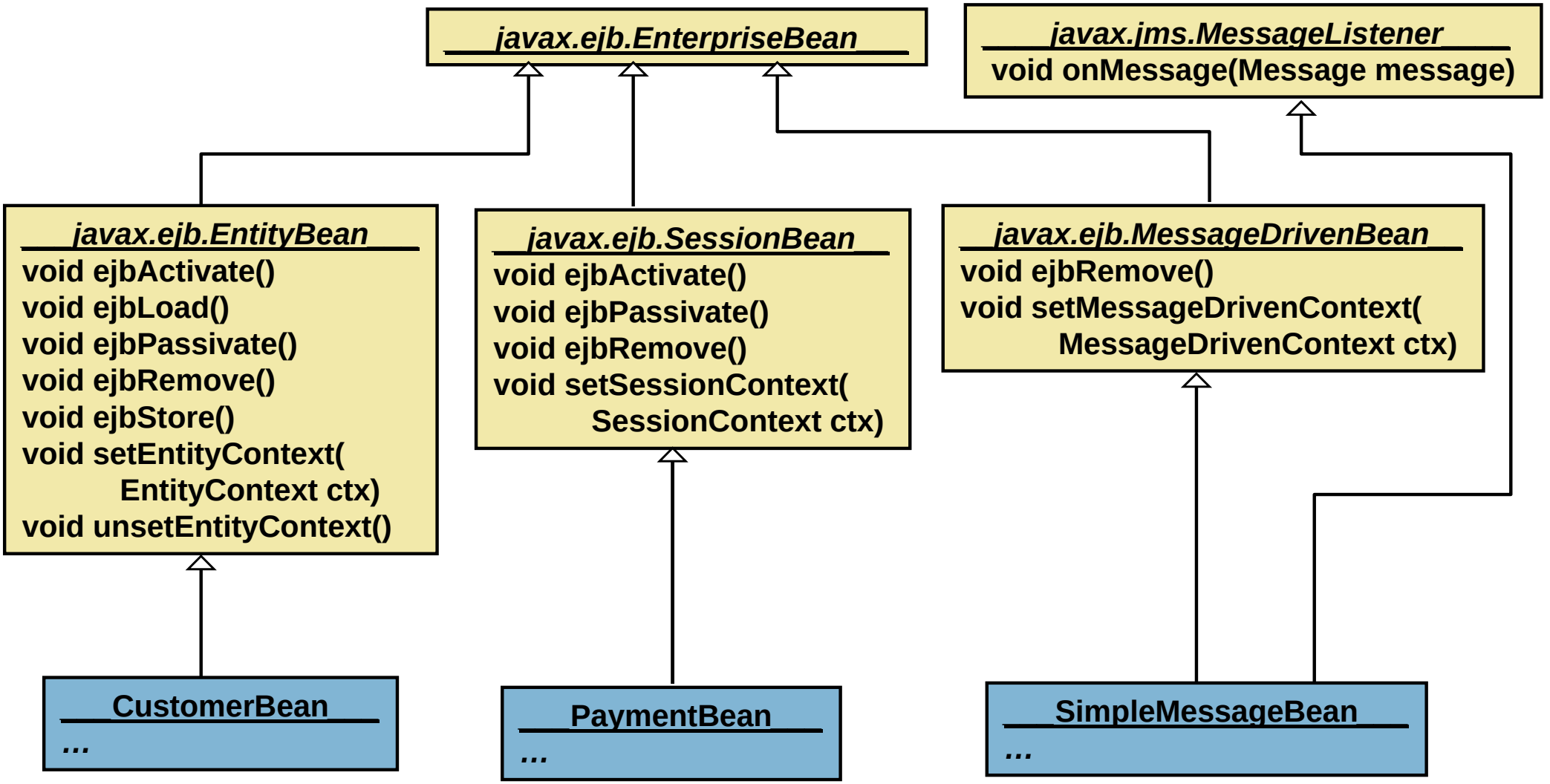
Example Bean Client

```
import javax.naming.*;
import java.util.*;

public class EJBClient {

  public static void main(String[] argv) throws Exception {

    Context ctx = new InitialContext();
    Object ref = ctx.lookup("CCPayment");

    PaymentHome home =
      (PaymentHome)PortableRemoteObject.narrow(
         ref, PaymentHome.class);

    Payment payment = home.create();

    payment.pay("1-2-3-4", "ulbi", "MastersCard",
                2009, 2, 47.11);

} }
```

EJB Interface Overview – Home and Remote



EJB Interface Overview – Bean Classes



Bibliography

- Sun Microsystems, Inc. *Enterprise JavaBeans Specification, Version 2.1*, 2003. **http://java.sun.com/products/ejb/**
- E. Armstrong, J. Ball, S. Bodoff, D. B. Carson, I. Evans, D. Green, K. Haase, and E. Jendrock. *The J2EE 1.4 Tutorial*, 2004. **http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html**
- R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 3rd edition, September 2001.

Fragen?