Java EE 5: Central Concepts

dr. gerald loeffler
enterprise software architect, shipserv ltd
gerald.loeffler@googlemail.com

Contents

- About this course
- Prerequisites
  - Java 5 features, dependency injection, AOP, layering
- Overview of Java EE 5
- The example application: TaskTracker
- Java Persistence API
- EJB 3 simplified API
- Implementing web services
- JavaServer Faces
- Asynchronous server-side Java
- Summary

Acronyms and abbreviations

- EJB – Enterprise JavaBean
  - SB – Session Bean
    - SFSB – StateFul Session Bean
    - SLSB – StateLess Session Bean
  - MDB – Message-Driven Bean
- JSF – JavaServer Faces
  - JSP – JavaServer Page
- JPA – Java Persistence API
- JAX-WS – Java API for XML Web Services
- IDE – Integrated Development Environment
- JSR – Java Specification Request

References

- JSR 220: Enterprise JavaBeans, Version 3.0
  - EJB 3.0 Simplified API, Proposed Final Draft, 18 Dec 2005, EJB 3.0 Expert Group, Sun Microsystems
  - Java Persistence API, Proposed Final Draft, 19 Dec 2005, EJB 3.0 Expert Group, Sun Microsystems
- JBoss documentation
  - JBoss EJB 3.0 Documentation
  - Embeddable EJB 3.0
  - JBoss Seam Documentation
- JavaServer Faces Specification, Version 1.2 Proposed Final Draft, Ed Burns, Roger Kitain (ed.), Sun Microsystems

About this course

Goals of this course

- Have a clear understanding of
  - The use of Java 5 annotations in Java EE 5
  - How to use Java EE 5 dependency injection
  - Options for layering Java EE applications
  - One recommended approach to implementing database-centric web applications with EJB 3 and JSF
  - How to implement web services with JAX-WS 2.0
- Have an approximate understanding of
  - AOP concepts and their use in Java EE
  - Asynchronous server-side architectures, JMS and MDBs
  - The purpose of most Java EE 5 technologies

Benotung

- Die Ermittlung der Note für diese KV erfolgt auf Basis einer schriftlichen multiple-choice Prüfung
  - Termin: siehe web-Seite der KV
  - Geprüft wird das Verstaendnis der Konzepte von Java EE und ihrer Zusammenhänge (und nicht ein enzyklopädisches Wissen über Programmierdetails)
    - Sie müssen JavaEE-Konzepte benennen, erklären und einordnen können.
    - Sie müssen nicht die exakten Namen von Java-Konstrukten (wie Java packages, classes, methods oder annotations) kennen, die in Java EE verwendet werden, aber es ist sehr wohl gefordert, über die prinzipielle Existenz und den Nutzen jener Java-Konstrukte Bescheid zu wissen, die in dieser KV besprochen werden.

Das Softwareentwicklungsprojekt

- …ist optional und wird nicht direkt benotet, ist aber ein wesentlicher Bestandteil dieser KV und sollte von allen Teilnehmern vor dem Antreten zur Prüfung durchgeführt werden.
  - Prüfungsfragen können sich auf das Softwareentwicklungsprojekt beziehen.
- Zu entwickeln ist:
  - Ein asynchroner Java EE job scheduler
  - Softwarearchitektur wie in dieser KV nahegelegt:
    - Web frontend mit JSF
    - Service und data access layer als EJB 3 session beans
    - Persistent domain objects als EJB 3 entities
    - Relationale Datenbank Ihrer Wahl

Das Softwareentwicklungsprojekt

- Persistent domain objects / database scheme etwa wie folgt:
  - JobClass:
    - name: id, "low"/"medium"/"high"
    - priority: integer, 1-3
  - Job:
    - id: integer, generated
    - jobClass: many-to-one to JobClass
    - userName
    - description
    - input: integer
    - output: integer
- Use cases die die Software unterstützen muss:
  - Submit job:
    - Im web frontend auswählen einer existierenden JobClass und Eingabe von description und input.

Das Softwareentwicklungsprojekt

- Das drücken des "submit"-buttons gibt unmittelbar Rückmeldung, dass der Job nun in Bearbeitung ist.
- Das Bearbeiten des Jobs, d.h. das Ermitteln des outputs (Ergebnisses) des Jobs auf Basis des inputs für den Job, geschieht asynchron. Der output eines Jobs ist immer

  output = 2*input
- Zugabe: "submit job" als SOAP web service exponieren.

– List processed jobs:
  - Im web frontend verfügbar.
  - Anzeige aller Jobs die für diesen user submitted wurden. Die Anzeige umfasst alle verfügbaren Daten zu einem Job, d.h. jedenfalls description und input, und, falls bereits verfügbar, auch den output der Berabeitung des jobs.
  - Zugabe: "list processed jobs" als SOAP web service exponieren.

Das Softwareentwicklungsprojekt

- Deliverables:
  - Ear
  - Beliebiger Automatismus zum Anlegen des Datenbank-Schemas (z.B. SQL-scripts, Ant script, Hibernate config)
  - SQL-scripts für Anlegen der JobClasses "low", "mdium" und "high".

Prerequisites

Java 5 annotations

- see examples

Java 5 generics

- see examples

Dependency injection

- Class A depends on the services provided by a class implementing interface S

```
public class A {
  private S s;
  public void m() {
    s.do();
  }
}
public interface S {
  void do();
}
```

Dependency injection

- Class A itself might instantiate an object of a class implementing interface S

```
public class A {
  private S s = new SImpl();
  //…
}
```

- Makes class A dependent on class SImpl

Dependency injection

- Class A might delegate instantiation to a factory

```
public class A {
  private S s = SFactory.createS();
  //…
}
```

- Makes class A dependent on factory class SFactory

Dependency injection

- Class A might expose its dependency on S through a setter and rely on "the container" to invoke that setter with an object implementing S

```
public class A {
  private S s;
  @Resource
  public void setS(S s) {
    this.s = s;
  }
  //…
}
```

Dependency injection

- Class A might expose its dependency on S through a field and rely on "the container" to set that field to an object implementing S

```
public class A {
  @Resource
  private S s;
  //…
}
```

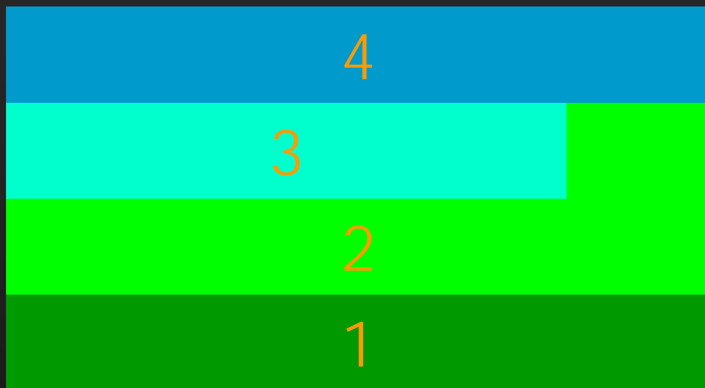
Aspect-oriented programming

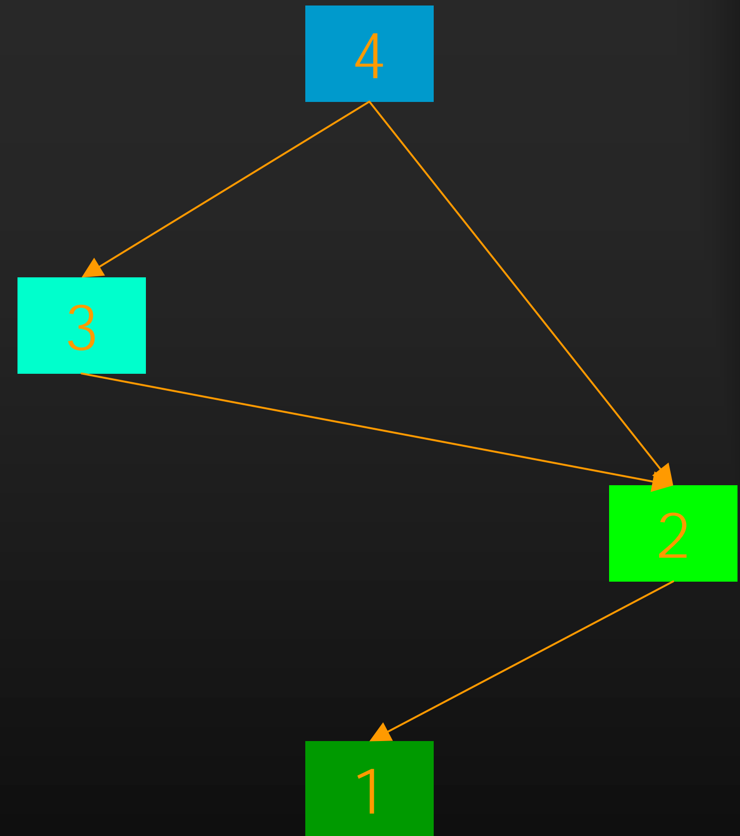
- Terminology and definitions:
  - Advice
  - Point-cut
  - Aspect
  - (Join point)
- Method interception
- Use cases:
  - logging, execution time measurements
  - security decisions
  - transaction management

Layering (in software design)

- Layering is a means of reducing the complexity (of software)
  - By reducing the number of interdependencies of software artefacts
- Layers are arranged in a stack
- Each layer depends only on the services offered (exposed) by the next lower layer(s)
  - ...and not on the services of the layers further up in the stack.
  - Every layer is ignorant of all higher layers.
- Layer is a logical concept
  - Tier is the corresponding physical concept

Layers (and tiers) form a stack





Layering server-side applications

- Typically, Java EE applications use 3 layers:
  - Presentation layer (GUI layer)
    - Handles user interaction
    - Rich-client GUI, web-based user interface (web-UI)
    - Often further structured according to Model-View-Controller (MVC)
  - Business logic layer (domain layer)
    - Business rules, domain logic, validation logic (often in addition to GUI), computation, workflow decisions
  - Data access layer (DAOs, persistence layer)
    - Handles access to and communication with back-end systems such as:
      - Relational databases (DBMSs)
      - Text-based indexing software (e.g. Lucene)
      - Message-oriented middleware (MOM, messaging systems)

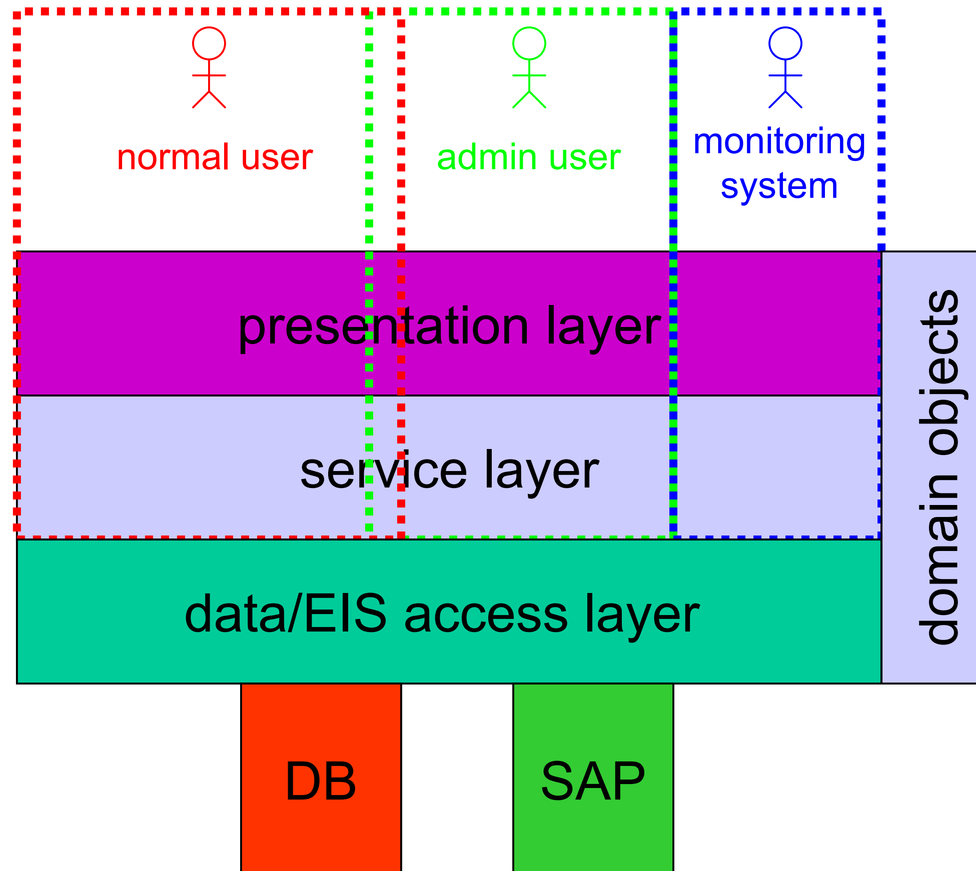
Layering server-side applications

    - Enterprise resource planning (ERP) systems (e.g. SAP)
  - Typically, these backend-systems reside in a separate tier
- Here we advocate additional layering decisions
  - Business logic layer is subdivided into
    - Service layer
      - Typically implements exactly what is needed by the presentation layer to support the required use cases
      - Can be stateful but is often stateless ("procedural")
      - Demarcates (begins/commits) transactions!
    - Domain objects (business object, domain model)
      - An object-oriented implementation of the concepts of the business domain, their behaviour and relationships.
      - Typically, the domain objects are persistent ("persistent domain model")
    - Domain objects are not a distinct layer because usually all other layers depend on the domain objects

Layering server-side applications

- The principles of layering imply
  - Service layer and data access layer are independent of presentation layer
  - Data access layer is independent of service layer
- As a convention, classes in the same layer belong to the same (root) Java package, e.g.
  - com.corp.proj.domain
  - com.corp.proj.dao
  - com.corp.proj.service
  - com.corp.proj.webui

Layering in J2EE applications 4/4



- Exercise: think of sensible assignments of these layers to tiers!
  - For different types of application: web app, standalone app

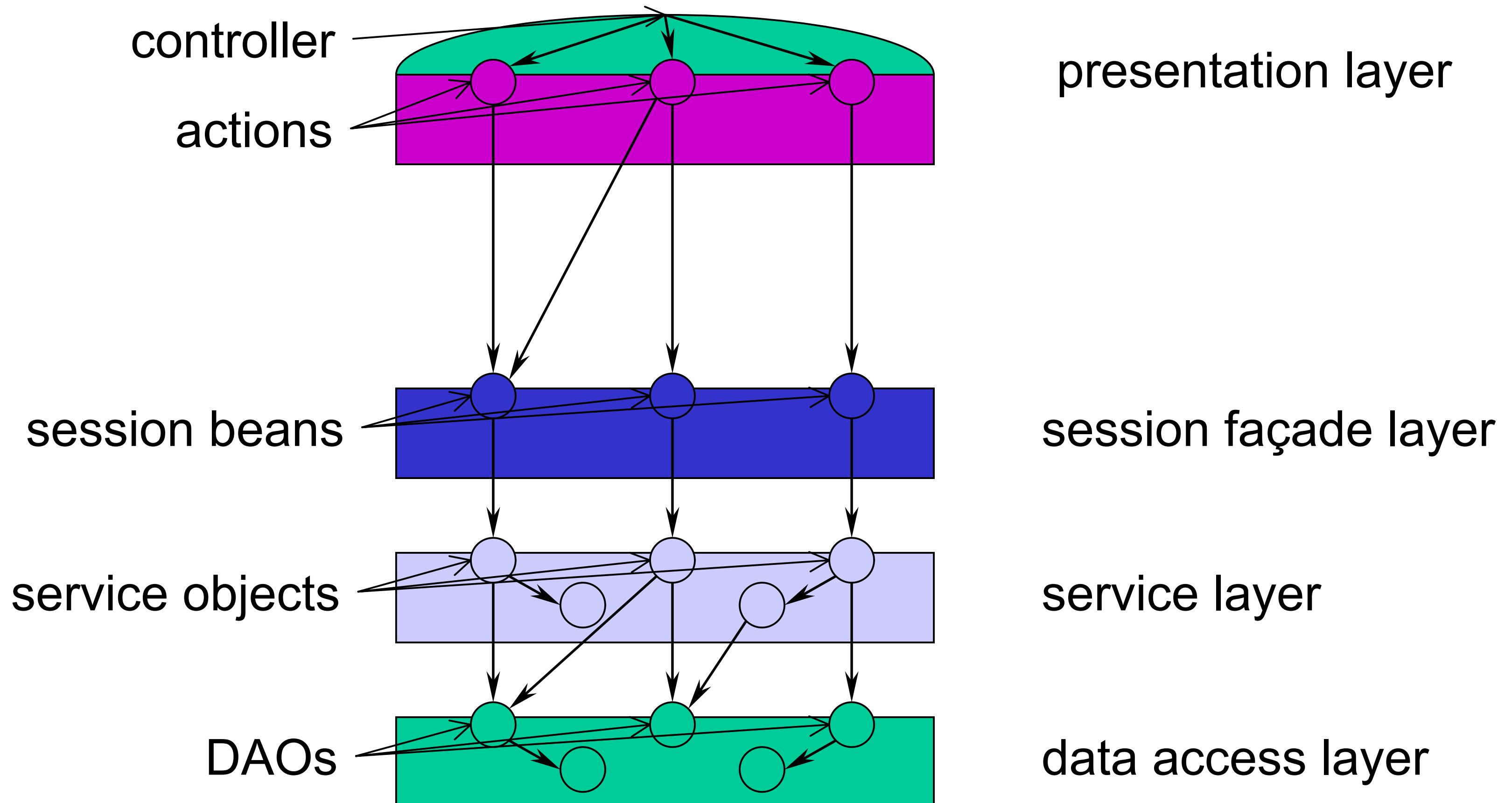
Other basic software design building blocks 1/2

- The classical GOF (Gamma et al.) design patterns
  - Singleton: Ensures a class only has one instance, and provide a global point of access to it
  - (Abstract) Factory: Provides an interface for creating families of related objects without specifying their concrete classes
  - Prototype: Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
  - Facade: Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that makes the subsystem easier to use.
  - Proxy: Provide a surrogate or placeholder for another object to control access to it
  - ...

Other basic software design building blocks 2/2

- Model-View-Controller (MVC) as a general design principle for user interfaces
- Distributed computing/J2EE/EJB patterns
  - Data Transfer Object: (Plain Java) classes which contain and encapsulate bulk data in one network transportable bundle
    - Domain Data Transfer Object vs. Custom Data Transfer Object
  - Session Facade: Clients should have access only to session beans (and not to entity beans)
  - EJBHomeFactory, BusinessDelegate, Business Interface: see later
- Very basic OO design principles
  - Encapsulation
  - Separation of concerns
  - ...

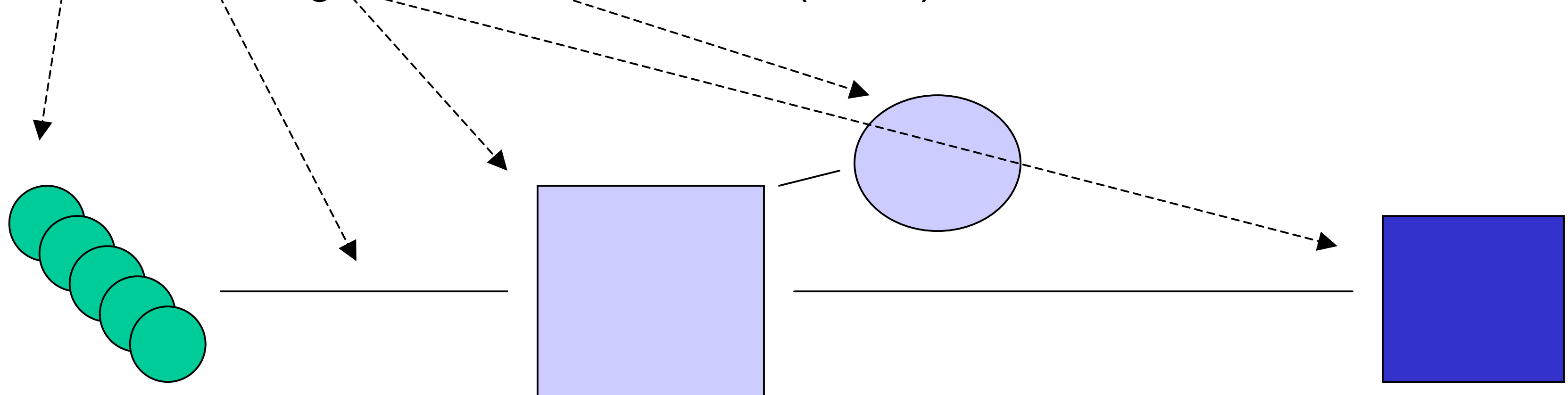
Spring and layering in J2EE applications 2/2



Overview of Java EE 5

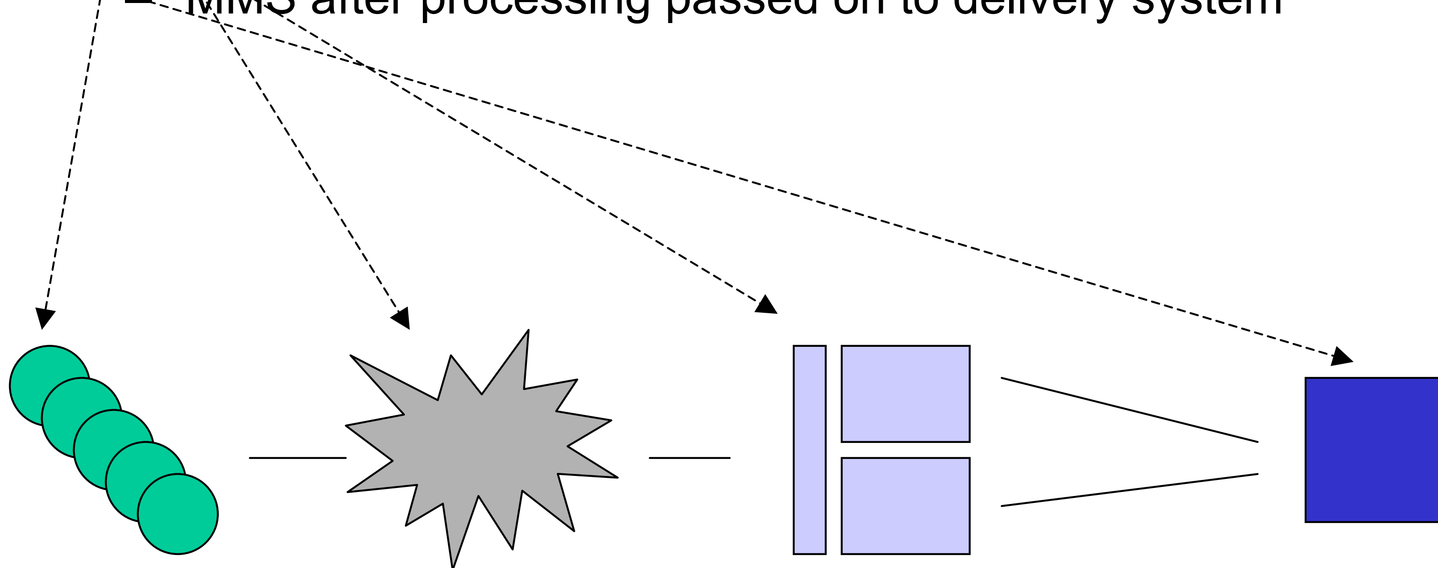
Examples of J2EE applications 1/3

- Portfolio Management System
  - intranet clients on bank-controlled PCs, globally; Swing frontend
  - one application server per continent
  - IIOP as protocol between clients and servers
  - database (data warehouse) to store portfolio information in
  - mainframes as back-end systems to handle trading (connected via message-oriented middleware (MOM)



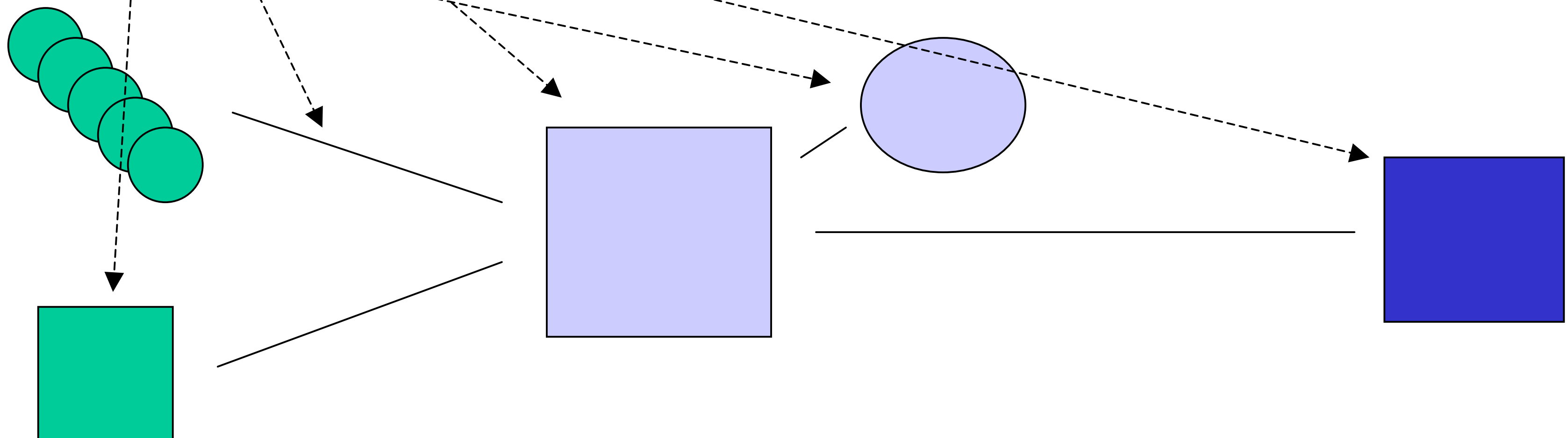
Examples of J2EE applications 2/3

- MMS processing system
  - MMS-capable phones as clients
  - mobile network infrastructure translates requests into HTTP
  - load-balanced application servers handle HTTP requests
  - MMS after processing passed on to delivery system



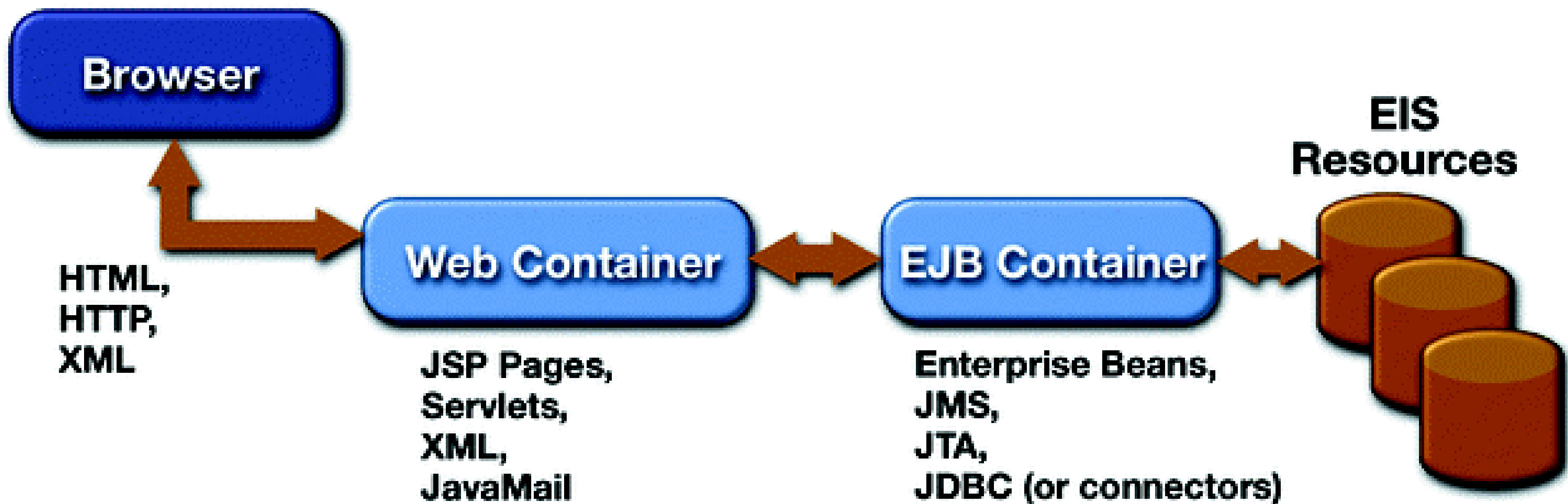
Examples of J2EE applications 3/3

- Mobile work management system
  - PDAs in factory setting as clients, receive work-units and confirm cmpletion; AWT-frontend
  - occasional HTTP-based connection to application server
  - web-application as control station to distribute work to clients
  - SAP back-end defines work-units and is notified of completion
  - database to "buffer" data between application core and SAP



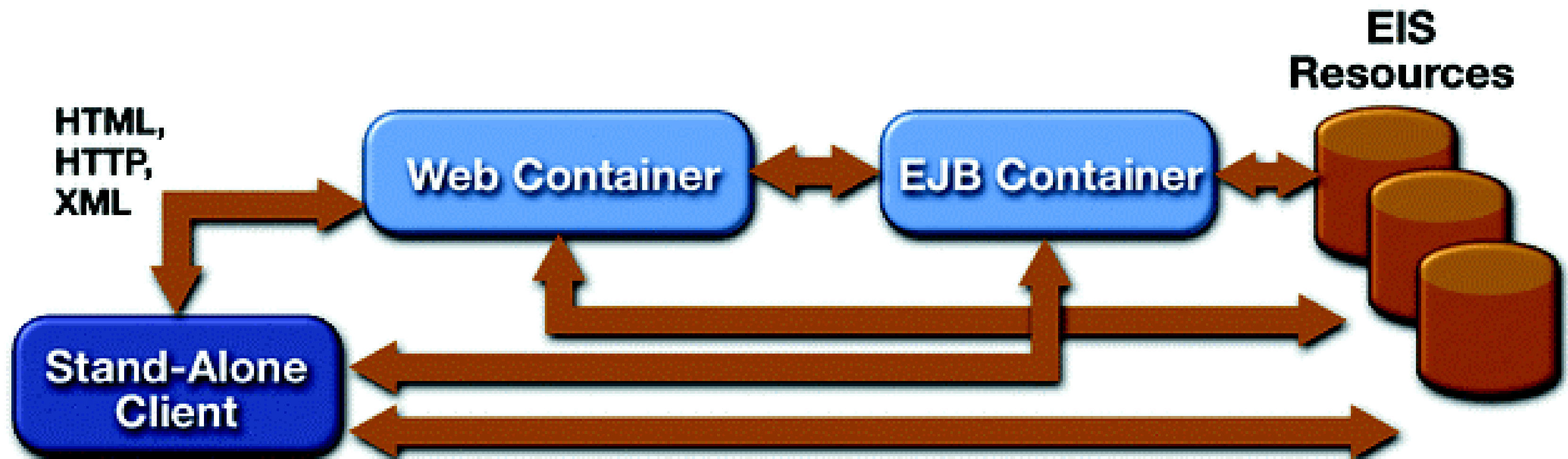
Examples of J2EE architectures 2/3

Web application using EJBs ("3.5 tier"), e.g. Task Tracker



Examples of J2EE architectures 3/3

Mixed-client (web service, Corba and DB) distributed application



Java EE is...

- ... Java Enterprise Edition
- ...what used to be called J2EE
- ...a collection of Java technologies to build distributed (incl. web) applications
- ...a set of specifications to write these applications against such that they are portable across application servers
- ...a set of specifications that defines the behaviour of application servers ("containers")
- ...an umbrella JSR (JSR TODO) and numerouse JSRs for the underlying technologies

Java EE 5 technologies / APIs

- EJB 3.0
- Servlet 2.5
- JSP 2.1
- JMS 1.1
- JTA 1.1
- JavaMail 1.4
- JAF 1.1
- Connector 1.5
- Web Services 1.2
- JAX-RPC 1.1
- JAX-WS 2.0

Java EE 5 technologies / APIs

- JAXB 2.0
- SAAJ 1.3
- JAXR 1.0
- Java EE Management 1.1
- Java EE Deployment 1.2
- JACC 1.1
- JSP Debugging 1.0
- JSTL 1.2
- Web Services Metadata 2.0
- JSF 1.2
- Common Annotations 1.0

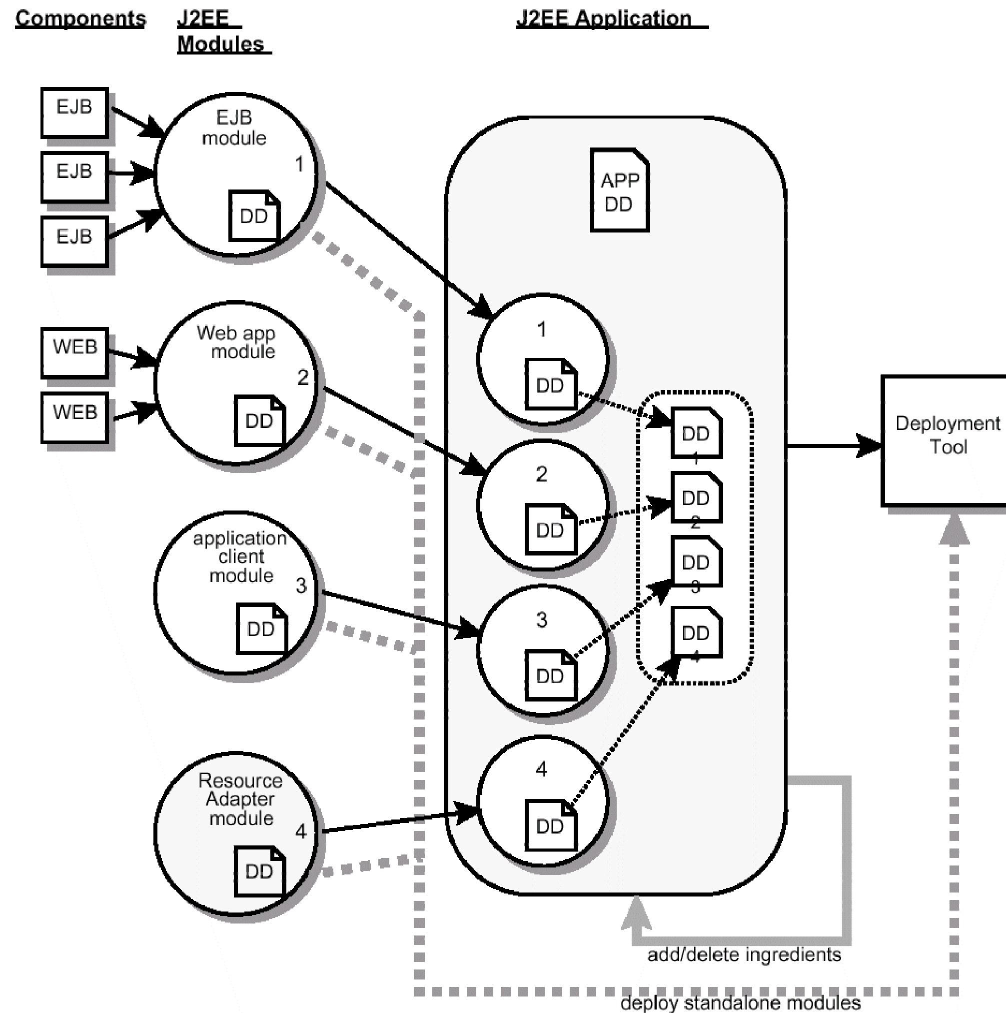
Java EE 5 technologies / APIs

- StAX 1.0
- Java Persistence 1.0
- all Java SE 5 APIs
  - Java IDL
  - JDBC
  - RMI-IIOP
  - JNDI
  - JAXP
  - JAAS
  - JMX

Java EE components

- Servlet
  - A Java class that consumes HTTP requests and produces a HTTP response for each request
  - Used by JSF
- JSP
  - "HTML file with Java code"
  - But can contain arbitrary markup (XML)
  - Typically used as a "view technology" in JSF
- EJB
  - Business logic component
- Resource adapter ("connector")
  - To plug an external, transactional system (resource; SAP, MOM, object cache) into an applicaction server
- Applet, application

Components, modules, deployment descriptors 1/3



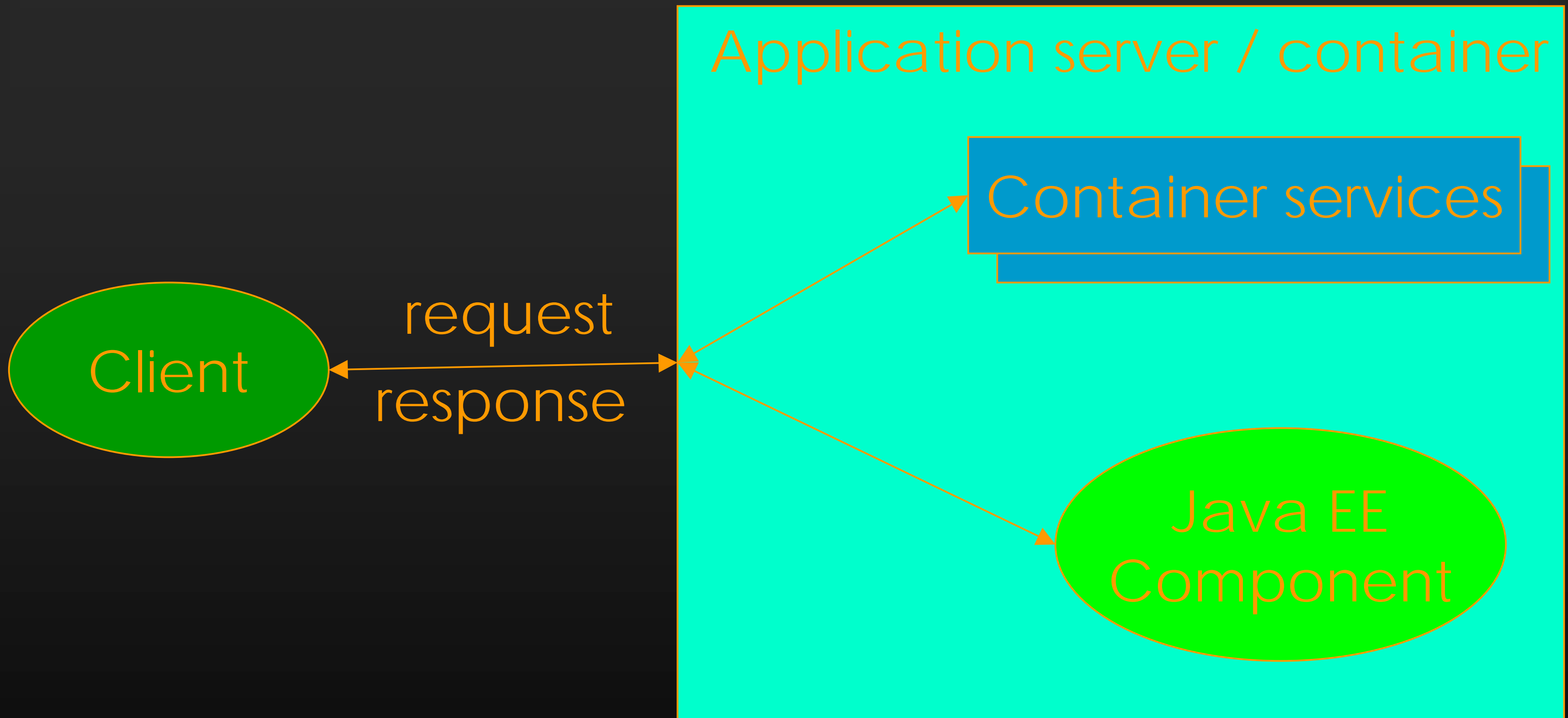
Components, modules, deployment descriptors 2/3

- Compononents are collections of Java classes/interfaces:
  - web: Servlet, JSP, ...
  - EJB: session bean, message driven bean, entity bean
  - ...
- Modules are jar-files that bundle components and follow a specific layout:
  - EJB module = ejb-jar file (xyz-ejb.jar)
    - DD: ejb-jar.xml (J2EE standard), sun-ejb-jar.xml (app server specific)
  - web app module = war (Web Archive) file (xyz.war)
    - DD: web.xml (J2EE standard), sun-web.xml (app server specific)
  - ...
- J2EE application (= ear (Enterprise Archive) file) is a jar-file that bundles other modules and follows a specific layout
  - DD: application.xml (J2EE standard), sun-application.xml (app server specific)

Application server

- Runtime environment for Java EE components
- Made up of containers
  - Web container hosts Servlets, JSPs, JSF applications
  - EJB container hosts EJBs
- Provides services to components
  - Dependency injection
  - Inerception
  - Thread pooling
  - State management
  - Security
    - Authentication, authorization (access control), encryption
  - Transaction management
- Is a web server, contains a transaction manager

Interception by the container



Java EE modules

- TODO: 43

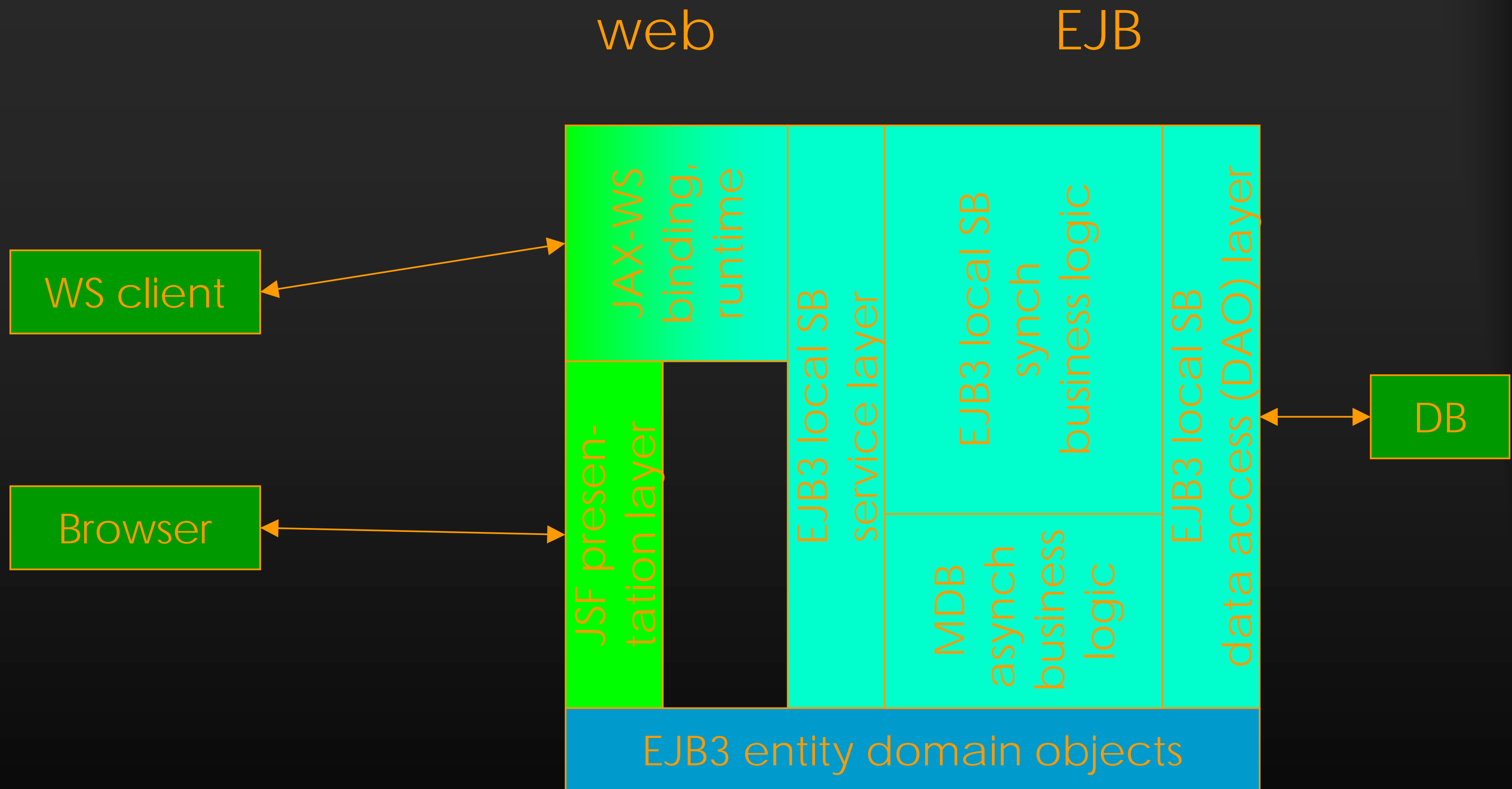
Software versions

- JBoss 4.0.4RC1
  - Includes EJB 3.0 container "on top of" J2EE 1.4
    - Includes snapshot of Hibernate for persistence
- Hibernate as an EJB 3 persistence provider
  - Hibernate Core
  - Hibernate Annotations
  - Hibernate Entity Manager
- Glassfish / Sun Java System App Server 9
  - Nascent reference implementation for Java EE 5
- Kodo 4.0.0 persistence provider early access
  - SolarMetric now owned by BEA
- JOnAS EJB 3 early preview

Prototypical software architecture

- For the most common Java EE scenario: a database-centric web application exposing some web services
  - Countless other scenarios and architectures possible
- Explicitly accounts for asynchronicity through message-driven beans

Prototypical software architecture

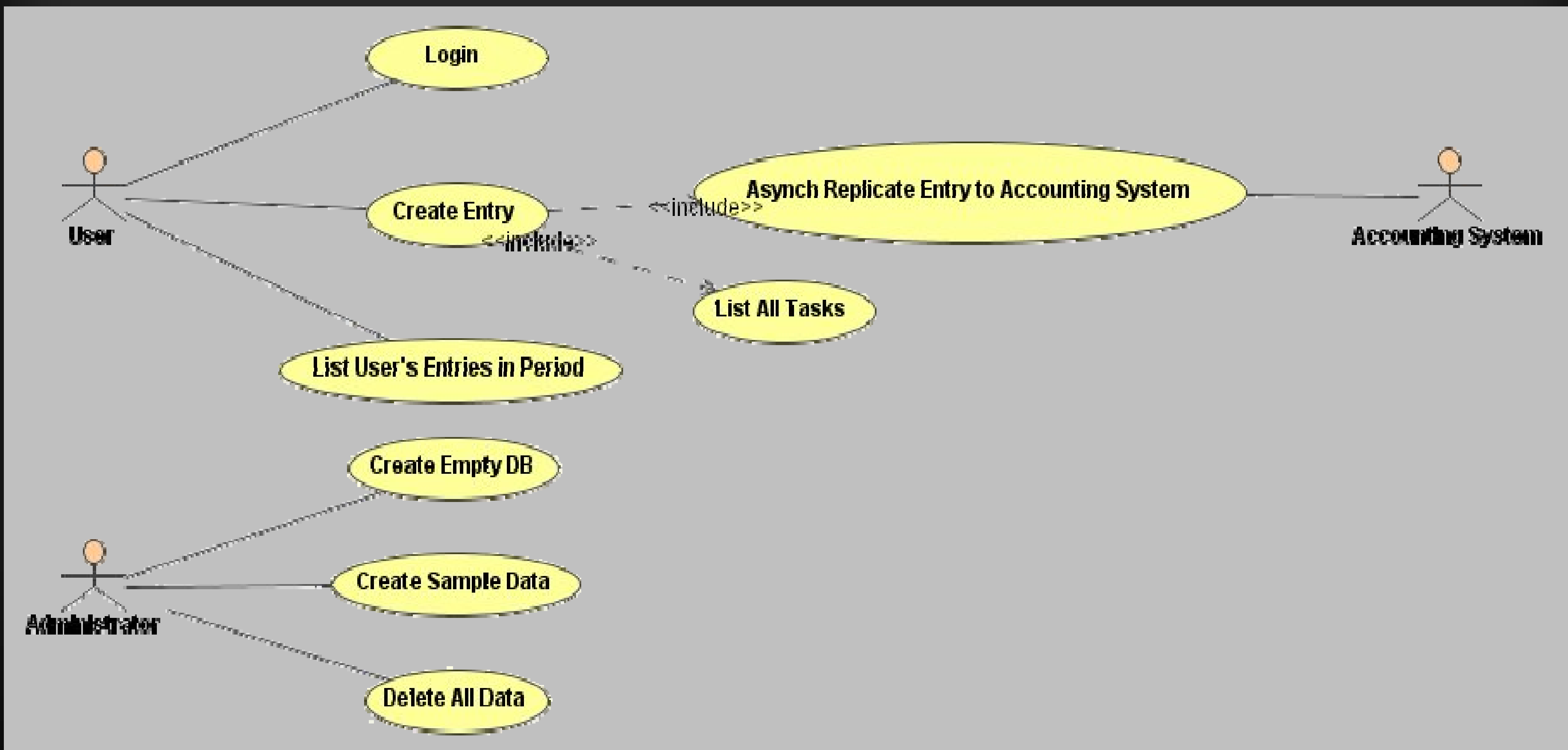


The example application: TaskTracker

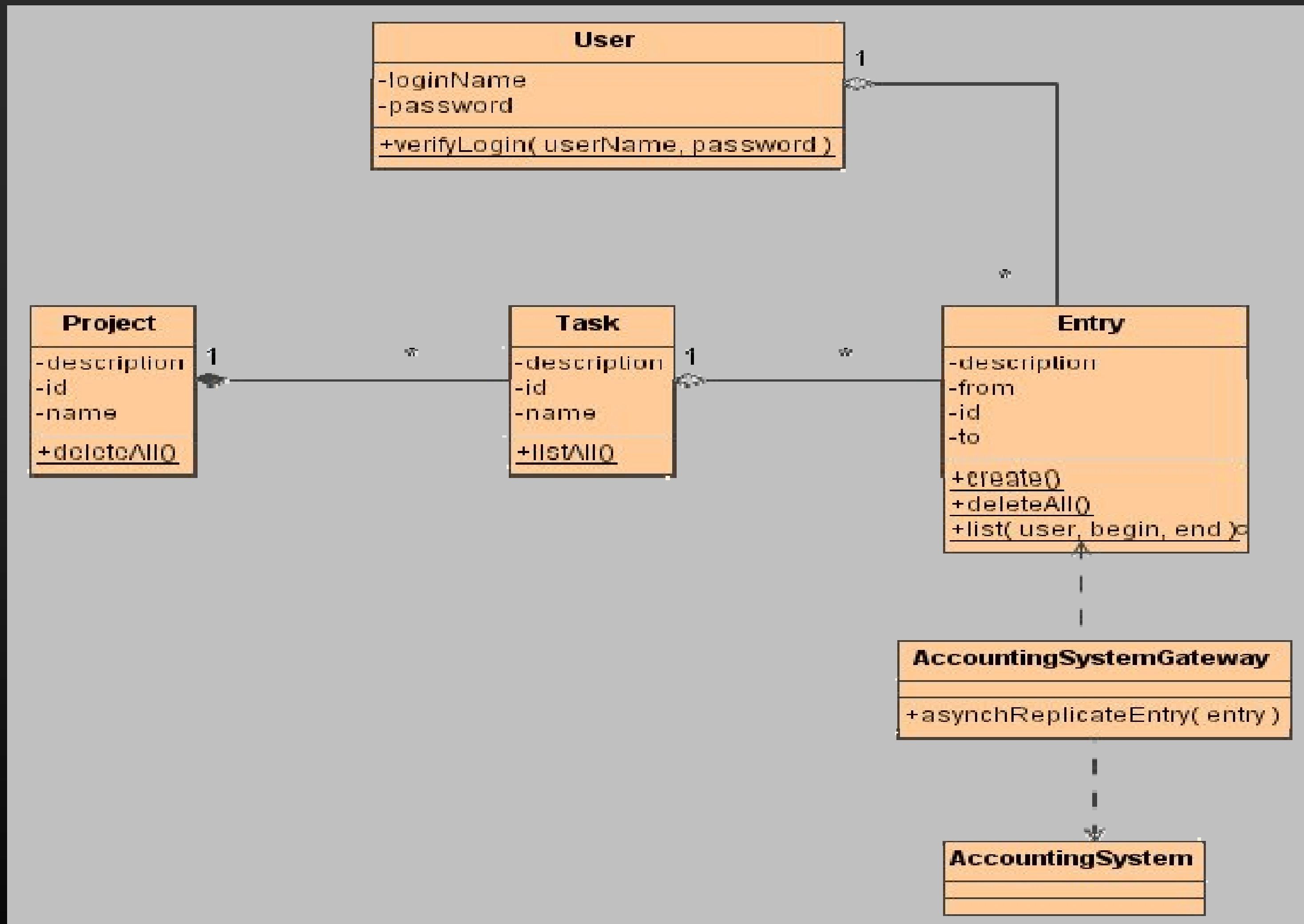
TaskTracker

- Is a simple database-centric web application
- Follows our “prototypical” software architecture
  - Has asynchronous business logic
  - Exposes web services in a Java-first manner
- Implemented on top of JBoss 4.0.4RC1
- Complete source code provided

TaskTracker use cases



TaskTracker domain object model

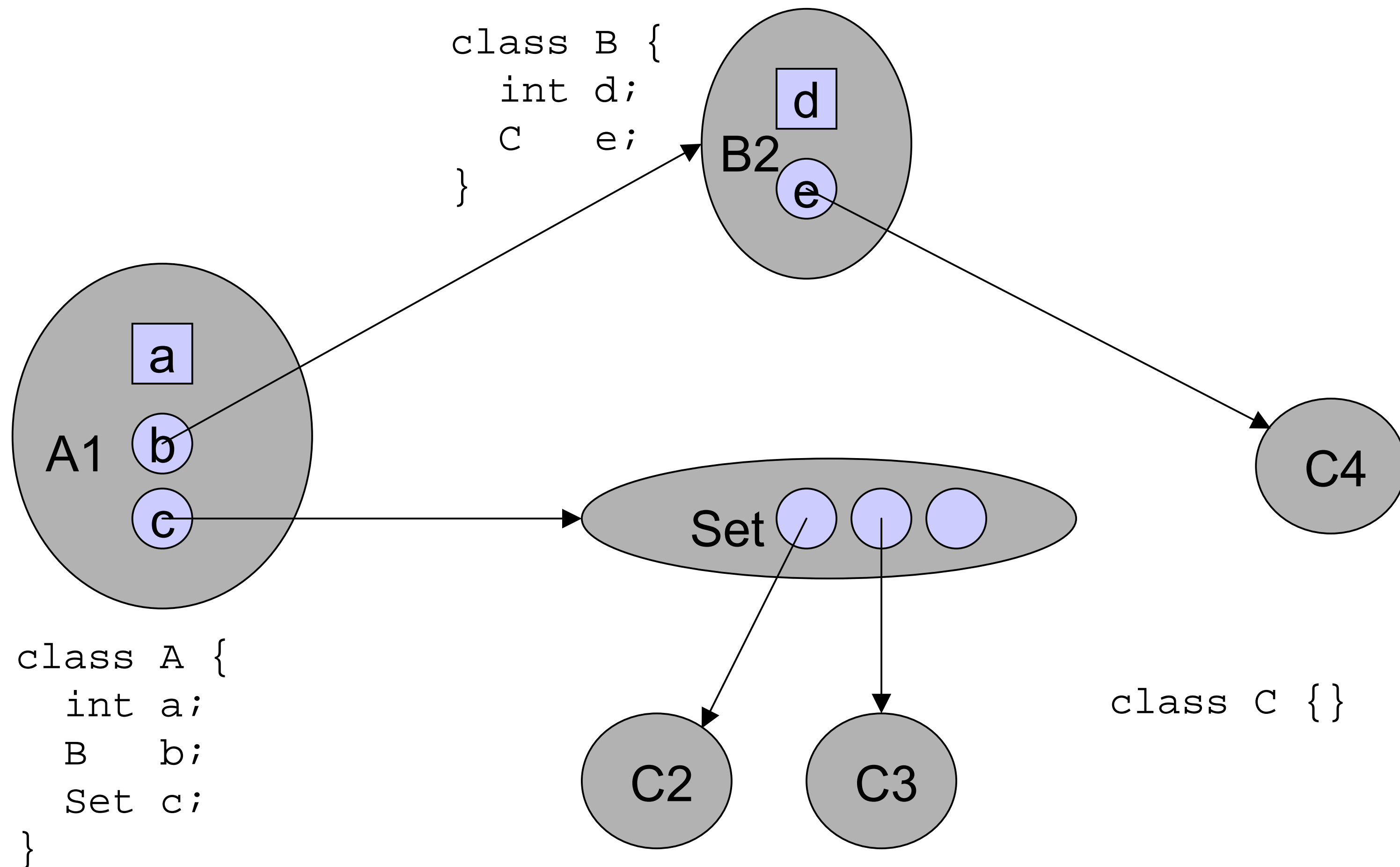


Java Persistence API

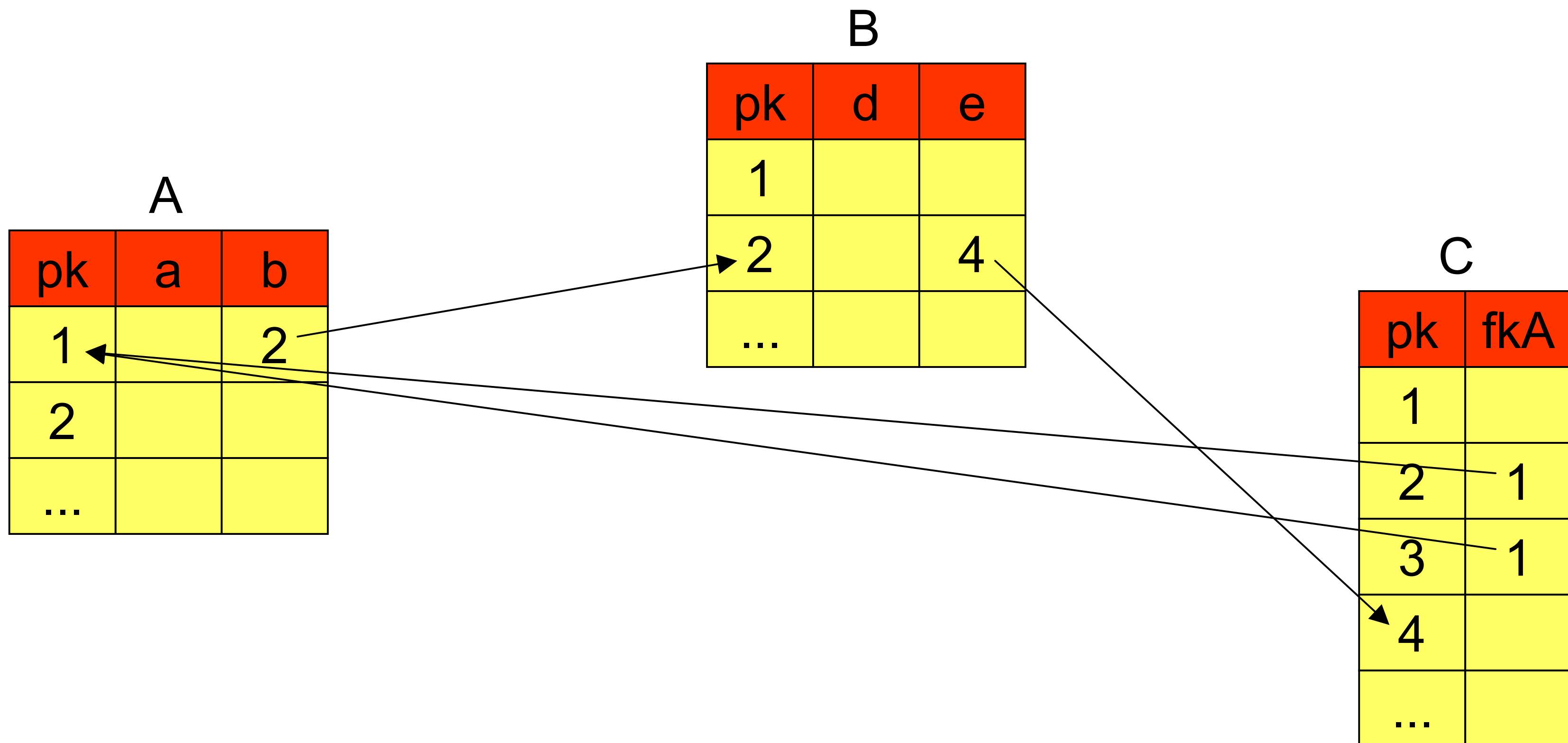
What it's about

- Object/relational mapping
- Focus is on (annotated) Java domain classes
  - Light-weight, local
- Query language is new version of EJB QL
- Can be used outside of EJB container
  - Web container
  - Java SE
- Very similar to Hibernate in "look&feel"
- Annotations and DDs supported
- Core concepts:
  - Persistence provider, Entity, EntityManager, persistence context, persistence.xml, orm.xml

Object/relational mapping (ORM) - object graph

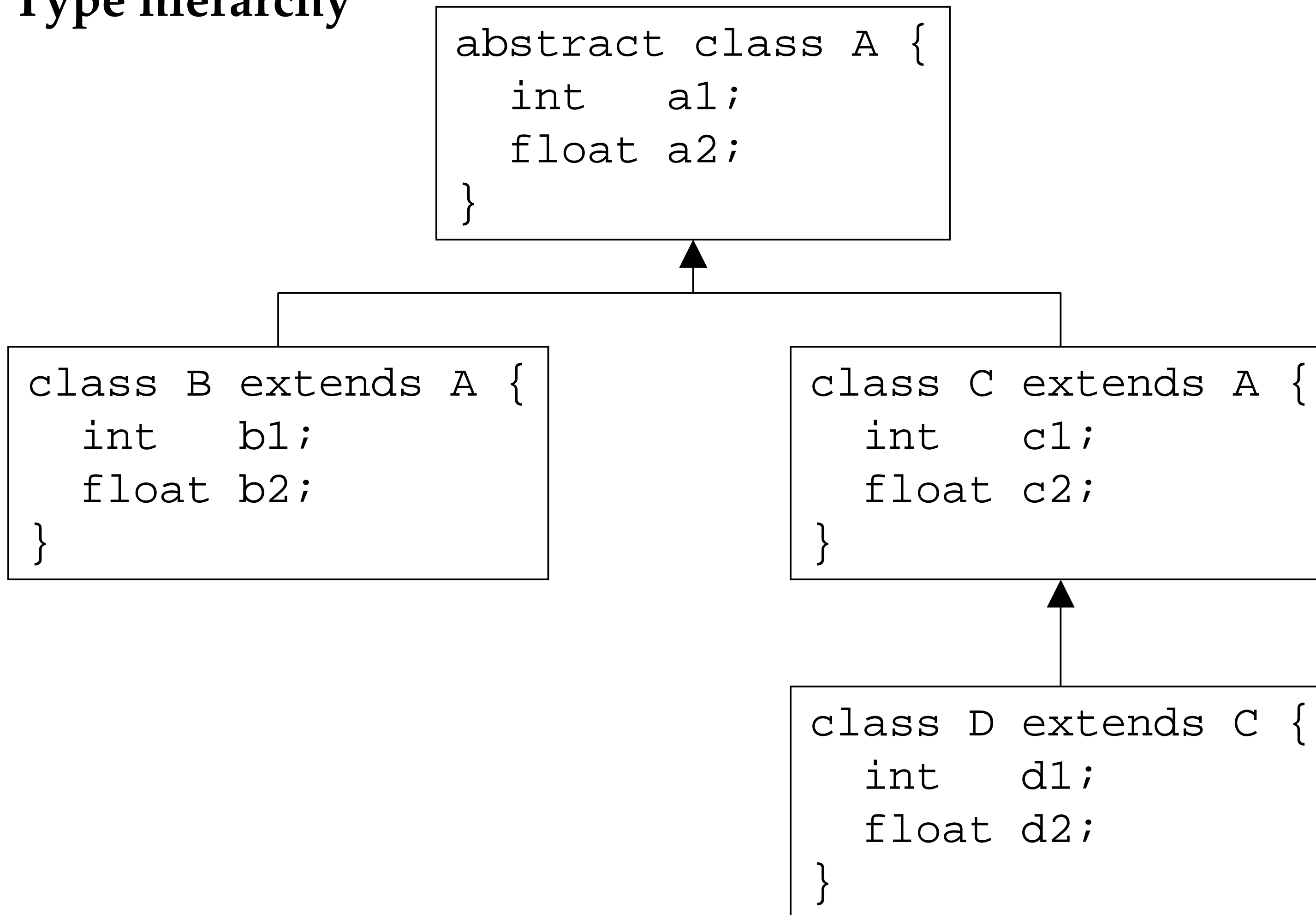


Object/relational mapping (ORM) - relations



ORM of inheritance hierarchies 1/4

Type hierarchy



ORM of inheritance hierarchies 2/4

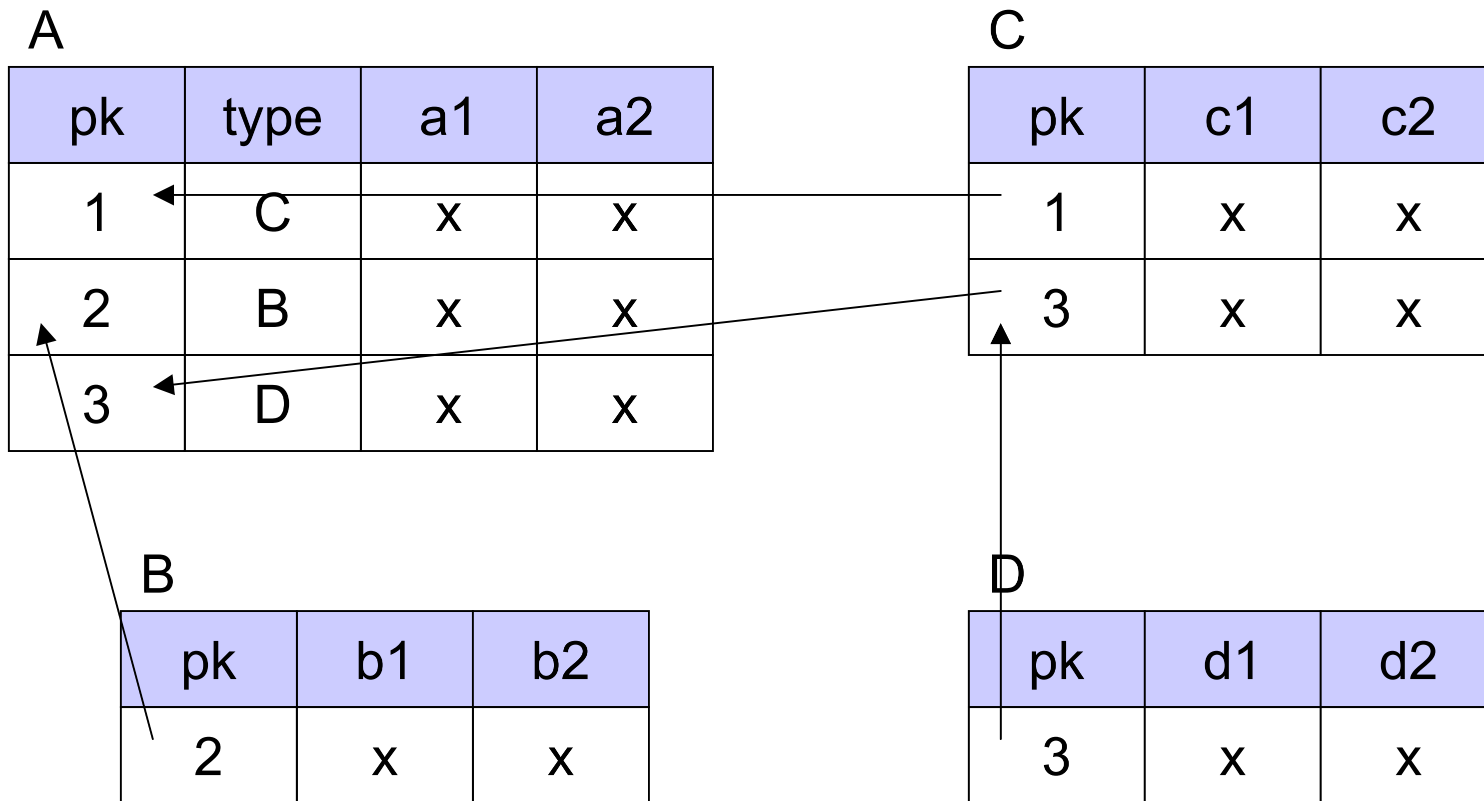
One table per type ("class") hierarchy

ABCD

pk	type	a1	a2	b1	b2	c1	c2	d1	d2
1	C	x	x			x	x		
2	B	x	x	x	x				
3	D	x	x			x	x	x	x

ORM of inheritance hierarchies 3/4

One table per type ("sub-class")



A

pk	type	a1	a2
1	C	x	x
2	B	x	x
3	D	x	x

B

pk	b1	b2
2	x	x

C

pk	c1	c2
1	x	x
3	x	x

D

pk	d1	d2
3	x	x

First a “realistic” example…

Family.java

```
// table: Family
// col: id primary key
// sequence: FAMILY_SEQU
@Entity
@SequenceGenerator(name="FamilySequ",
  sequenceName="FAMILY_SEQU")
public class Family {
  @Id
  @GeneratedValue(strategy=GenerationType.
    SEQUENCE, generator="FamilySequ")
  private Integer id;
  // inverse side of bidirectional 1-to-many
  @OneToMany(mappedBy="family", cascade=ALL)
  private Set<FamilyMember> members;
}
```

FamilyMember.java

```
// table: FAMILY_MEMBER
// col: id primary key
// col: family_id references FAMILY.id
@Entity
@Table(name = "FAMILY_MEMBER")
@Inheritance(strategy = InheritanceType.JOINED)
public class FamilyMember {
  @Id
  private Integer id;
  // owning side of birectional many-to-1
  @ManyToOne(cascade = { CascadeType.MERGE,
      CascadeType.PERSIST, CascadeType.REFRESH
  })
  private Family family;
}
```

Parent.java

```java
// table: Parent
// col: id primary key refs FAMILY_MEMBER.id
@Entity
public class Parent extends FamilyMember {
  // ...
}
```

Child.java

```
// table: Child
// col: id primary key references FAMILY_MEMBER.id
// col: reatreat_id unique refs PRIV_CHILD_ROOM.id
// join table: Child_Toy
// col: Child_id references Child.id
// col: toys_name references Toy.name
@Entity
@NamedQuery(name="findAChild",
  query="select c from Child c where...")
public class Child extends FamilyMember {
   // owning side of bidirectional 1-to-1
   @OneToOne(cascade = { MERGE, PERSIST, REFRESH })
   private PrivateChildRoom retreat;
   // (owning side of) unidirectional many-to-many
   @ManyToMany(cascade = { MERGE, PERSIST, REFRESH })
   private Set<Toy> toys = new LinkedHashSet<Toy>();
}
```

PrivateChildRoom.java

```java
// table: PRIV_CHILD_ROOM
// col: id primary key
@Entity
@Table(name = "PRIV_CHILD_ROOM")
public class PrivateChildRoom {
  @Id
  private Integer id;
  @OneToOne(mappedBy = "retreat", cascade = {
      CascadeType.MERGE, CascadeType.PERSIST,
      CascadeType.REFRESH })
  // inverse side of bidirectional 1-to-1
  private Child occupiedBy;
}
```

Toy.java

```
// table: Toy
// col: name varchar2(255) primary key
public class Toy {
  @Id
  private String name;
}
```

…now the theory

Entities

- `@Entity` annotation
- Public/protected no-arg constructor
- Not inner class
- Not final, no methods final
- Can be `Serializable`
  - E.g. if used as a Data Transfer Object
- Can be abstract
- Can be part of inheritance tree

Persistent state and access to it

- Access to fields
  - By persistence provider
    - Property-based: through JavaBeans property accessors
      - Annotate getter (!)
      - Accessors must be public or protected
      - Exclusion from persistent state: `@Transient`
    - Field-based: direct
      - Annotate fields
      - Exclusion from persistent state: `transient` or `@Transient`
  - By clients of entity: only through accessors
    - Fields must not be public
    - (And should really be private)
- Personal recommendation:
  - Use field-based access

Persistent state

- Types of fields/properties
  - Primitives, primitive wrappers, `String`
  - `BigInteger`, `BigDecimal`
  - `Date`, `Calendar`
  - `java.sql`-types: `Date`, `Time`, `Timestamp`
  - `Char[]`, `Character[]`
  - `Byte[]`, `Byte[]`
  - `Collection` (sub-) interfaces
    - `Collection`, `Set`, `List`, `Map`
  - `Enums`
  - `Entities`
  - `Embeddables`

Entity identity: primary keys

- Every identity has an id, corresponds to primary key in the database
  - Is immutable after `persist()`
- Simple primary key
  - Single field mapped to single column: `@Id`
- Composite primary key
  - Primary key class with multiple fields mapped to multiple columns
    - Either embedded in entity: `@EmbeddedId`
    - Or referenced from entity and primary key fields duplicated in entity: `@IdClass`
    - Public no-arg constructor, `Serializable`, `equals()` and `hashCode()` based on primary key columns

Family.java

```
// table: Family
// col: id primary key
// sequence: FAMILY_SEQU
@Entity
@SequenceGenerator(name="FamilySequ“,
  sequenceName="FAMILY_SEQU“
public class Family {
  @Id
  @GeneratedValue(strategy=GenerationType.
    SEQUENCE, generator="FamilySequ")
  private Integer id;
  // inverse side of bidirectional 1-to-many
  @OneToMany(mappedBy="family", cascade=ALL)
  private Set<FamilyMember> members;
}
```

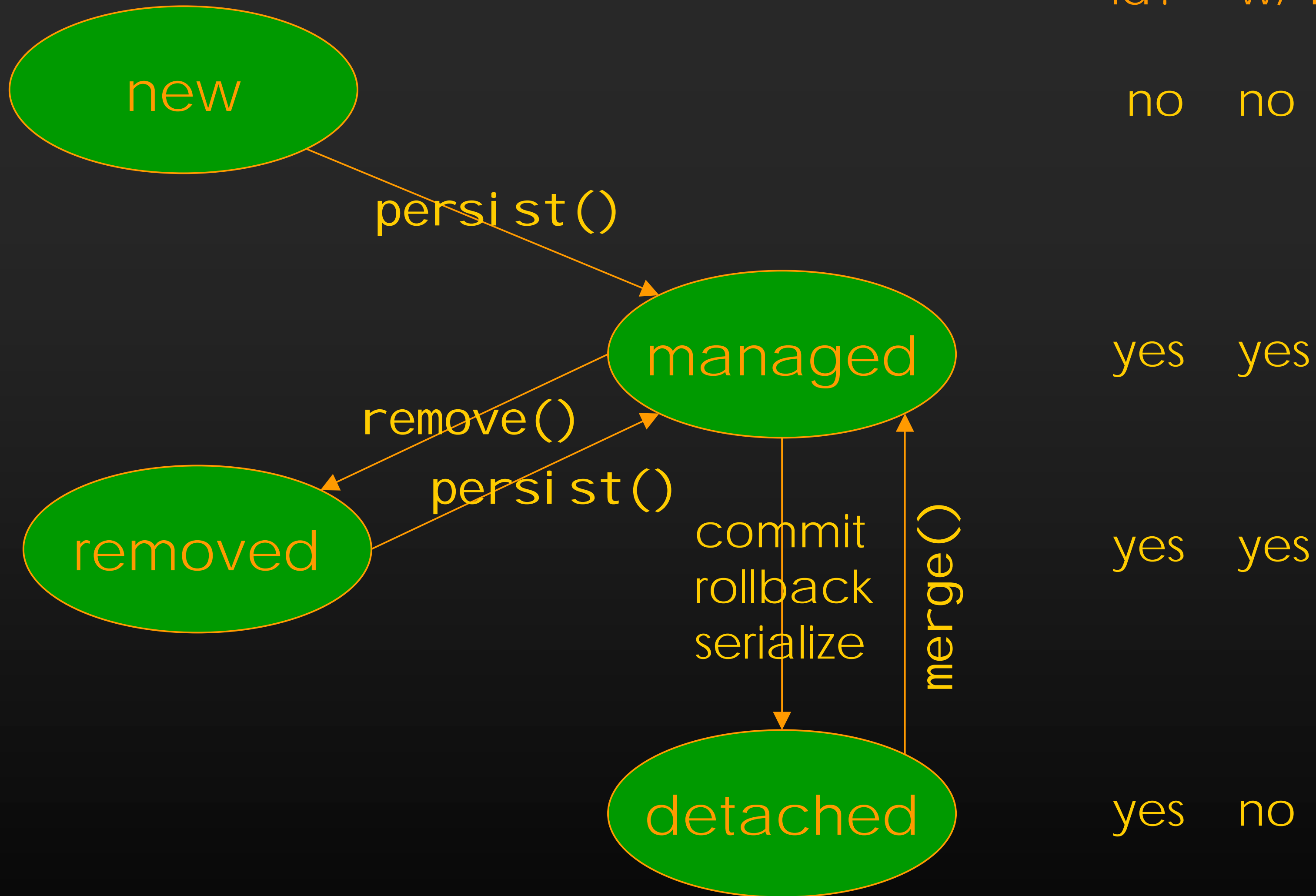
Persistence context

- Accessed through `EntityManager`
- Comprises a set of entity instances where at most one instance exists for each persistent entity identity (= primary key value)
  - Is exactly what you expect: "one row of a table corresponds to one object"
- Persistence context lifetime
  - Defined when EntityManager is created (injected)
  - Transaction-scoped persistence context
    - Default and most natural
  - Extended persistence context
    - PC spans more than one transaction

EntityManager

- Works on exactly one persistence context
  - Defines the API to interact with persistence context
- Important operations:
  - `void persist(Object entityInstance)`
  - `void remove(Object entityInstance)`
  - `T merge (T entityInstance)`
    - Detached object support
  - `T find(Class<T> entity, Object id)`
  - `void refresh(Object entityInstance)`
  - `Query createNamedQuery(String queryName)`
    - EJB QL query string defined with `@NamedQuery`
    - SQL query string defined with `@NamedNativeQuery`

Entity states



Cascading entity state changes

- Each entity-entity relationship may be annotated with one or more `CascadeType`s:
  - Default is to not cascade!
  - `CascadeType`: `PERSIST`, `REMOVE`, `MERGE`, `REFRESH`, `ALL` to propagate the corresponding state changes (method calls) on the entity instance w/ the annotated reference to the referenced entity instance(s).
  - No delete-orphan!
  - REMOVE not portable for ManyToOne or ManyToMany
- A note on `CascadeType.MERGE`
  - As `merge()` can result in a copy of the instance to be returned, cascading `merge()` requires that the reference be re-set to that copy. This is done by the persistence provider.
  - Can thus result in a completely new object graph!

Synchronisation with database

- Flush: entity instance state written to database
  - Only effects entities currently associated with PC
  - Simplest case: automatic flush at commit
  - Explicitly through `EntityManager.flush()`
  - Typically (Hibernate) before query execution
    - Ensure that entity state always obeys all constraints
- Refresh: database to entity instance state
  - Only explicitly for each instance through `EntityManager.refresh()`
  - Overwrites un-flushed changes to entity instance

Entity manager management

- Container-managed entity manager (Java EE)
  - = transaction-propagated persistence context
    - Persistence context flows with JTA transactions
      - All `EntityManager`s invoked within a transaction access the same persistence context
  - Lifecycle (create, close) of EntityManager is managed by container and persistence provider
  - If EntityManager is injected or looked up via JNDI
    - Happens automatically
- Application-managed entity manager (EE, SE)
  - = stand-alone persistence context
    - Persistence context does not flow with JTA transactions
  - Lifecycle of EntityManager is managed by application
    - `EntityManagerFactory`, `EntityManager.close()`

FamilyService.java

```
public interface FamilyService {
  Family createRandomFamily()
}
```

FamilyServiceImpl1.java

```
// container-managed entity manager
// transaction-scoped persistence context
@Stateless
public class FamilyServiceImpl1 implements
    FamilyService {
  @PersistenceContext
  private EntityManager em;

  public Family createRandomFamily() {
    Family f = new Family();
    Child c = (Child) em.createNamedQuery(
        "findAChild").getSingleResult();
    // ..
    em.persist(f);
    return f; // f now detached
  }
}
```

FamilyServiceImpl3.java (1/2)

```
// application-managed entity manager
// transaction-scoped persistence context
@Stateless
public class FamilyServiceImpl3 implements
    FamilyService {
  @PersistenceUnit
  private EntityManagerFactory emf;
  private EntityManager em;

  @PostConstruct
  public void createEM() {
    em = emf.createEntityManager();
  }
```

FamilyServiceImpl3.java (2/2)

```
  @PreDestroy
  public void closeEM() {
    em.close();
  }
  public Family createRandomFamily() {
    Family f = new Family();
    Child c = (Child) em.createNamedQuery(
                   "findAChild").getSingleResult();
    // ..
    em.persist(f);
    return f; // f now detached
  }
}
```

Extended persistence context

- Within container only in stateful session beans
  - `@PersistenceContext(type=EXTENDED)`
- PC exists from creation of `EntityManager` to close
  - For container-managed: from time of injection or JNDI-lookup until execution of `@Remove` method
  - For application-managed: from creation using `EntityManagerFactory` to call of `EntityManager.close()`
- PC may span several transactions
  - `EntityManager` transparently participates in the transactions in which it is invoked
  - Entity instances remain associated with the PC after transaction commit/rollback
    - Do not become detached

Extended persistence context

- Allows PC operations to be called outside a transaction
  - `persist(), remove(), merge(), refresh()`
- No more exceptions when accessing lazily initialised relationships, no need to `merge()`
- Restrictions for container-managed entity managers (transaction-propagated persistence contexts) apply
  - Call from session bean with transcation-scoped PC to session bean with extended PC within the same JTA transaction raises exception
- Two orthogonal concepts: lifetime and propagation behaviour of persistence context!

FamilyServiceConversational.java

```
public interface FamilyServiceConversational
    extends FamilyService {
  void goAway();
}
```

FamilyServiceImpl2.java

```
// container-managed entity manager
// extended persistence context
@Stateful
public class FamilyServiceImpl2 implements
    FamilyServiceConversational {
  @PersistenceContext(type=EXTENDED)
  private EntityManager em;
  private Family f;
  public Family createRandomFamily() {
    f = new Family();
    Child c = (Child) em.createNamedQuery(
        "findAChild").getSingleResult();
    // ..
    em.persist(f);
    return f; // f still managed
  }
  @Remove
  public void goAway() {}
}
```

FamilyServiceImpl4.java (1/2)

```
// application-managed entity manager
// extended persistence context
@Stateful
public class FamilyServiceImpl4 implements
    FamilyServiceConversational {
  @PersistenceUnit
  private EntityManagerFactory emf;
  private EntityManager em;
  private Family f;

  @PostConstruct
  public void createEM() {
    em = emf.createEntityManager(
         PersistenceContextType.EXTENDED);
  }
```

FamilyServiceImpl4.java (2/2)

```
  public Family createRandomFamily() {
    f = new Family();
    Child c = (Child) em.createNamedQuery(
                 "findAChild").getSingleResult();
    // ..
    em.persist(f);
    return f; // f still managed
  }
  @Remove
  public void goAway() {
    em.close(); // f now detached
  }
}
```

Embeddable

- A class that isn't an entity itself but exists only as part of an entity
  - No id
  - It's state is mapped to columns of its entity
    - Using the same access type as its entity
  - Not shareable between between entities (classes)
- Used to map coarse-grained database schema to fine-grained object model
- @Embeddabl e on class and/or @Embedded on field/property of its entity

Entity relationships

- If an entity references one or more other entities
  - One-to-one, one-to-many, many-to-one, many-to-many
  - A relationship to entity A can hold entity instances of sub-classes of A (polymorphism, just as in Java itself)
- Java type of the referenced entity must be known: generics or annotation parameter
- Fetch strategy: eager vs. lazy
  - Defines *when* a relationship is loaded from the database – not *how*
  - Defaults: eager for *ToOne, lazy for *ToMany
  - Entity instance must be associated with PC for fetch…

Family1.java

```
// default mapping, insecure bidirectional rels
@Entity
public class Family1 {
  @Id
  private Integer id;
  @OneToMany(mappedBy = "family")
  private Set<Child1> children;

  public Set<Child1> getChildren() {
    return children;
  }
  public void setChildren(Set<Child1> children) {
    this.children = children;
  }
}
```

Child1.java

```
@Entity
public class Child1 {
  @Id
  private Integer id;
  @ManyToOne(fetch=FetchType.LAZY)
  private Family1 family;

  public Family1 getFamily() {return family;}
  public void setFamily(Family1 family) {
    this.family = family;
  }
}
```

Bidirectional entity relationships

- A references B and B references A
- Have owning and inverse side
  - Owning side determines foreign key value!
  - Inverse side may be followed (cascade) but is not used to set the foreign key value!
  - Mapping annotations must be on owning side
- Many-to-one and one-to-many:
  - Many-side is owner (because it contains the foreign key)
  - One-side is always inverse side
- One-to-one:
  - Side containing the foreign key is the owning side
  - Specify **mappedBy** on inverse side

Bidirectional entity relationships

- Many-to-many:
  - Mapped through join table (with 2 foreign keys)
  - Denote arbitrary side as inverse using **mappedBy** to refer to field/property on owning side
- Persistence runtime does not maintain referential integrity of bidirectional relationships!
  - Include required logic in accessors
    - An argument for field-based access
    - Would be cleanest to disallow direct mutable access to Collections of many-valued relationships
      - Implement add/remove methods instead
  - Might use helper-library like Gemini instead

Family2.java

```
// manually secured bidirectional rel
@Entity
public class Family2 {
  @Id private Integer id;
  @OneToMany(mappedBy = "family")
  private Set<Child2> children=new HashSet<Child2>();
  /** do not alter returned Set */
  public Set<Child2> getChildren() {return children;}
  // no setter for children
  public void addChild(Child2 child) {
    if (child != null) child.setFamily(this);
  }
  public void removeChild(Child2 child) {
    if (child != null && children.contains(child))
      child.setFamily(null);
  }
}
```

Child2.java

```
// manually secured bidirectional rels
@Entity
public class Child2 {
   @Id
   private Integer id;
   @ManyToOne
   private Family2 family;

   public Family2 getFamily() {return family;}
   public void setFamily(Family2 family) {
     if (this.family != null &&
          !this.family.equals(family)) {
       this.family.getChildren().remove(this);
     }
     this.family = family;
     if (family!= null) family.getChildren().add(this);
   }
}
```

Family3.java

```
// referential integrity maintained by Gemini
@Entity
public class Family3 {
  @Id
  private Integer id;
  @OneToMany(mappedBy = "family")
  @BidirectionalMany(oppositeName = "family",
    initOnlyFirstTime = true)
  private Set<Child3> children;

  public Set<Child3> getChildren() {
    return children;
  }
  public void setChildren(Set<Child3> children) {
    this.children = children;
  }
}
```

Child3.java

```
// referential integrity maintained by Gemini
@Entity
public class Child3 {
   @Id
   private Integer id;
   @ManyToOne
   @BidirectionalOne(oppositeName="children",
     oppositeType = BidirectionalMany.class)
   private Family3 family;

   public Family3 getFamily() {return family;}
   public void setFamily(Family3 family) {
     this.family = family;
   }
}
```

Mapping defaults

- Define how relational database schema is derived from (annotated) domain model
  - All defaults can be overridden with annotations or in orm.xml
- Entity name -> table name
- Field/property name -> column name
- Join table
  - For many-to-many or unidirectional one-to-many (!)
  - Name: <entity1>_<entity2>
- Foreign key column name:
  - <field>_<pk> if field/property name is available
  - <entity>_<pk> otherwise
    - foreign key in join table referencing owning side in an uni-directional one-to-many or many-to-many relationship

Inheritance mapping

- Three strategies defined:
  - Single table per class hierarchy
    - Complete inheritance graph collapsed into one table
    - All state of subclasses must map to nullable columns
  - Single table per concrete class (optional)
  - Joined subclass
    - Each class has its table, with primary key acting as foreign key into table of superclass
      - Thank god for single inheritance in Java
    - Most clean and flexible, may be slow due to joins
- Joined subclass strategy

FamilyMember.java

```
// table: FAMILY_MEMBER
// col: id primary key
// col: family_id references FAMILY.id
@Entity
@Table(name = "FAMILY_MEMBER")
@Inheritance(strategy = InheritanceType.JOINED)
public class FamilyMember {
   @Id
   private Integer id;
   // owning side of birectional many-to-1
   @ManyToOne(cascade = { CascadeType.MERGE,
       CascadeType.PERSIST, CascadeType.REFRESH
   })
   private Family family;
}
```

Parent.java

```
// table: Parent
// col: id primary key refs FAMILY_MEMBER.id
@Entity
public class Parent extends FamilyMember {
  // ...
}
```

Child.java

```
// table: Child
// col: id primary key references FAMILY_MEMBER.id
// col: reatreat_id unique refs PRIV_CHILD_ROOM.id
// join table: Child_Toy
// col: Child_id references Child.id
// col: toys_name references Toy.name
@Entity
@NamedQuery(name="findAChild",
  query="select c from Child c where...")
public class Child extends FamilyMember {
   // owning side of bidirectional 1-to-1
   @OneToOne(cascade = { MERGE, PERSIST, REFRESH })
   private PrivateChildRoom retreat;
   // (owning side of) unidirectional many-to-many
   @ManyToMany(cascade = { MERGE, PERSIST, REFRESH })
   private Set<Toy> toys = new LinkedHashSet<Toy>();
}
```

Persistence unit and packaging

- Persistence unit comprises
  - META-INF/persistence.xml
    - Defines persistence unit and its name
    - The persistence provider (e.g. Hibernate)
    - The DataSource JNDI name
    - Think hibernate.cfg.xml
  - Compiled entities
  - Optional META-INF/orm.xml
- No new archive (no "par"), may be contained in
  - EJB-JAR
  - WAR
  - JAR which may be contained in
    - WAR, EAR, application-client-jar

What we didn’t talk about

- The query language EJB QL
- Most Id generation strategies
- Details of composite primary keys
- Callbacks
- Use of FlushMode
- Optimistic locking, version fields
- Native queries and SQL result set mapping
- Details of mapping annotations
- persistence.xml and orm.xml
- Details of use in Java SE environment

EJB 3 simplified API

Themes

- Simplifying the developer's task
- Metadata annotations in addition to XML
- Configuration by exception
- Session beans: no required home interface
- Entity beans: light-weight ORM
- Interceptors for session beans and MDBs
- Architectural properties
  - of session beans and MDBs remain essentially unchanged
  - of EJB 3 entities are very similar to Hibernate entities

Session and message-driven beans

- Focus on (annotated) enterprise bean class
- Deployment descriptors available
  - To override annotations
  - May be sparse
- Support interception of invocation of
  - Business methods
  - Lifecycle callbacks
- Can be target of dependency injection
- Transaction management:
  - Default is container-managed
  - `@TransactionManagement` on bean class to specify bean-managed
  - `@TransactionAttribute` on bean or business methods

FamilyService1.java

```
// plain interface
public interface FamilyService1 {
  Object createNewFamily(String spec);
}
```

FamilyServiceImpl1.java

```
// stateless, local, CMT, two interceptors
// name defaults to FamilyServiceImpl1
@Stateless
@Interceptors( { LogInterceptor.class })
public class FamilyServiceImpl1 implements
     FamilyService1 {
   public Object createNewFamily(String spec) {
     Object family = null;
     // ...
    return family;
   }
   @AroundInvoke
   public Object spy(InvocationContext ictx)
       throws Exception {
     // tell the neighbours...
     return ictx.proceed();
   }
}
```

LogInterceptor.java

```
public class LogInterceptor {
  @AroundInvoke
  public Object log(InvocationContext ictx)
      throws Exception {
    try {
      // log args
      Object ret = ictx.proceed();
      // log return value
      return ret;
    } catch (Exception e) {
      // log exception
      throw e;
    }
  }
}
```

Lifecycle and lifecycle callbacks

1. Instantiation of bean instance
2. Dependency injection
3. `@PostConstruct`
   - Unspecified transaction and security context
4. Invocations of business methods
5. Invocation of business method with @Remove
   - Stateful session beans only
6. `@PreDestroy`
   - Unspecified transaction and security context
7. Destruction of bean instance

Lifecycle and lifecycle callbacks

- For stateful session beans only: between `@PostConstruct` and `@PreDestroy` possibly arbitrary pairs of
  - `@PrePassivate`
  - Passivation (serialization) to external storage
  - Activation (deserialization) from external storage
  - `@PostActivate`

Session beans

- Business interface is plain Java interface
  - No required component interface (`EJB(Local)Object`)
- Default is local business interface
  - Use `@Local` and `@Remote`
    - If more than one interface implemented
    - If a remote interface is required
    - May appear on business interface or bean class
  - `Serializable`, `Exteranlizable` can't be business interfaces
- Declare arbitrary exceptions on business methods
  - Don't use `RemoteException`
  - Remote client sees `EJBException`

Session beans

- No need for home interface
  - Lookup (incl. injection) returns EJB instance

FamilyService2.java

```
@Remote
public interface FamilyService2 {
  void initFamily(String mothersName);
  void addChild(String name);
  void enough() throws NotYetEnough;
}
```

WorkService2.java

```java
// plain interface
public interface WorkService2 {
  double earnIncome();
}
```

FamilyServiceImpl2.java

```
// stateful, remote and local, CMT
@Stateful(name = "StatefulFamilyService")
@Local( { WorkService2.class })
public class FamilyServiceImpl2 implements
    FamilyService2, WorkService2 {
  public void initFamily(String mothersName) {}
  public void addChild(String name) {}
  @Remove(retainIfException = true)
  public void enough() throws NotYetEnough {}
  @TransactionAttribute(
    TransactionAttributeType.REQUIRES_NEW)
  public double earnIncome() {
    return 0.0;
  }
}
```

Stateless session beans

- `@Stateless` annotation
  - No need to implement `SessionBean`
- Lifecycle callbacks
  - `@PostConstruct`, `@PreDestroy`
- `@WebService`, `@WebMethod` for definition of web services (JSR 181)

Stateful session beans

- `@Stateful` annotation
  - No need to implement `SessionBean` or `Serializable`
- Implementation of `SessionSynchronization` supported
- Lifecycle callbacks
  - `@PostConstruct`, `@PreDestroy`
  - `@PostActivate`, `@PrePassivate`
- Lookup returns new instance
  - Typically needs initialization via business method(s)!
- `@Remove` annotates a “normal” business method
  - Client initiates removal by calling this method
  - Removal through container after completion

Interceptor methods

- Specialised AOP facility
  - Only around advice
  - Only on session beans and MDBs
    - Only business methods and lifecycle callbacks
- Method with `@AroundInvoke`
  - Only one per bean class
  - `public Object *(InvocationContext) throws Exception`
  - `InvocationContext` passed around as data holder
  - `InvocationContext.proceed()` to proceed
- Become "part of" method invocation
  - Share transaction and security context
  - Can throw same exceptions as "their" method
  - Can invoke JNDI, JDBC, JMS, EJBs, Entity Manager

FamilyServiceImpl1.java

```
// stateless, local, CMT, two interceptors
// name defaults to FamilyServiceImpl1
@Stateless
@Interceptors( { LogInterceptor.class })
public class FamilyServiceImpl1 implements
    FamilyService1 {
  public Object createNewFamily(String spec) {
    Object family = null;
    // ...
    return family;
  }
  @AroundInvoke
  public Object spy(InvocationContext ictx)
      throws Exception {
    // tell the neighbours...
    return ictx.proceed();
  }
}
```

Interceptor classes

- Holds interceptor method
  - Same rules as for those
  - Only one `@AroundInvoke` per interceptor class
- Stateless, associated with enterprise bean
  - `InvocationContext` passed around as data holder
- Public no-arg constructor
- Definition and assocation with beans is static
  - Default interceptors
    - Apply to all session beans and MDBs in ejb-jar
    - Defined in deployment descriptor
  - Denoted on bean using `@Interceptors`
    - Definition order is invocation order
- Can be target of dependency injection

LogInterceptor.java

```
public class LogInterceptor {
  @AroundInvoke
  public Object log(InvocationContext ictx)
      throws Exception {
    try {
      // log args
      Object ret = ictx.proceed();
      // log return value
      return ret;
    } catch (Exception e) {
      // log exception
      throw e;
    }
  }
}
```

Dependency injection

- Injection of
  - `EJBContext` (`SessionContext`) (first)
  - `DataSource`
  - `UserTransaction`
  - `EntityManager`
  - Anything (?) that can be looked up via JNDI in `java:comp/env`
- Injection into fields or with setters
  - `@EJB`, `@Resource`
  - Missing information inferred from field or setter
    - Resource type
      - From field/property type
    - Resource name
      - From field/property name

FamilyService3.java

```
public interface FamilyService3 {
  Object generateBoyfriend(String spec);
}
```

FamilyServiceImpl3.java

```
// declarative security, EJB ref, resource ref
@Stateless
@DeclareRoles( { "Adolescent", "MidlifeCrisis" })
@RunAs("Adolescent")
public class FamilyServiceImpl3 implements
    FamilyService3 {
  @EJB
  private FamilyService1 consequences;
  @Resource(name = "jms/NappyQ")
  private QueueConnectionFactory qcf;
  @RolesAllowed( { "Adolescent“,"MidlifeCrisis" })
  public Object generateBoyfriend(String spec) {
    Object boy = null;
    // ...
    Object f = consequences.createNewFamily(spec);
    // ...
    return boy;
  }
}
```

Transaction fundamentals 1/4

- A transaction is a *unit of work* that has the ACID properties
  - atomicity:either the complete unit of work is performed or none at all - all or nothing
  - consistency:by executing the unit of work, a system is transferred from one consistent state to another consistent state, irrespective of whether the transaction succeeds or fails
  - isolation:the effects of a transaction are invisible to other transactions as long as the (original) transaction has not succeeded (cf. transaction isolation level)
  - durability: the effect of a transaction is (usually) persistent and survives system failures/shutdowns
- Local transaction vs. global (distributed) transaction
  - a local transaction involves exactly one transactional resource, e.g. a relational database
  - a distributed transaction involves several transactional resources, e.g. a relational database and a messaging system (JMS provider)

Transaction fundamentals 2/4

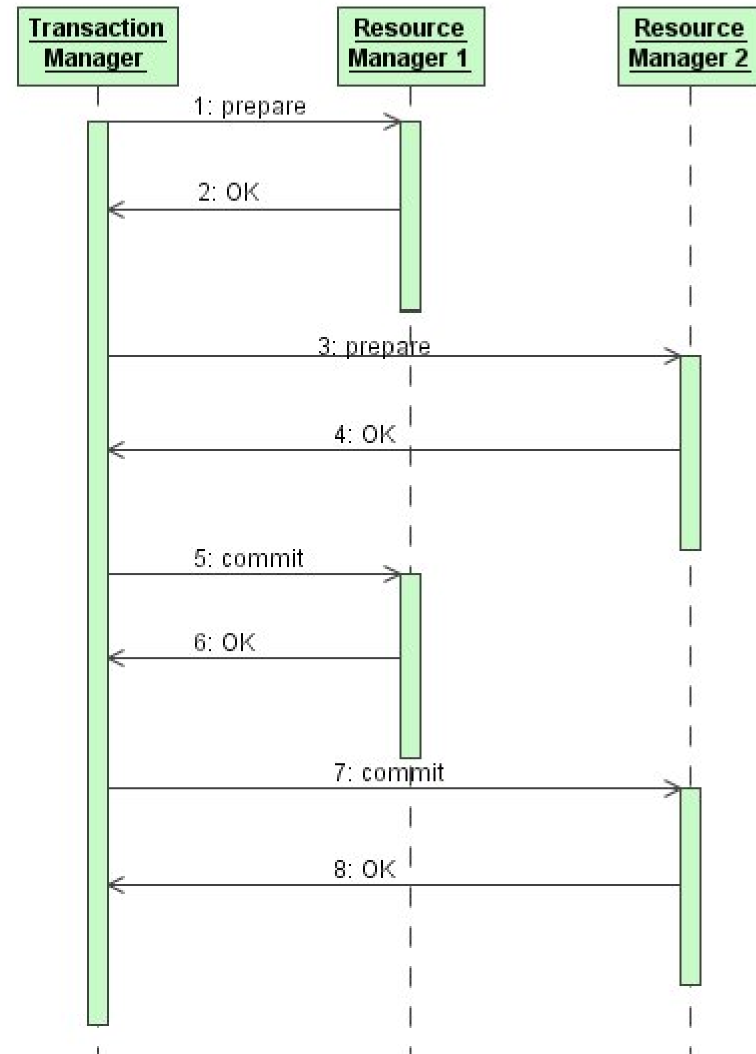
- Participants in a distributed transaction:
  - application: uses the facilities of the application server to begin/commit/rollback transactions, which in turn delegates this responsibility to the transaction manager
  - application server: uses a transaction manager (which is usually part of the application server) to coordinate transactions by calling begin(), commit(), etc. on the transaction manager
  - transaction manager: coordinates transactions across several transactional resources by enlisting them and orchestrating a two-phase commit protocol among them
  - resource (manager): the resource manager is the entity that interacts with the transaction manager on behalf of a transactional resource, e.g. a RDBMS would be a transactional resource and the JDBC driver would be the resource manager for that resource

Transaction fundamentals 3/4

- The Distributed Transaction Processing (DTP) model of the Open Group (formerly X/Open) defines interfaces between the basic components of a distributed transaction system:
  - TX is the interface that a transaction manager exposes to the application or application server
    - begin(), commit(), rollback()
  - XA is the (bidirectional) interface between a resource manager and a transaction manager
    - e.g. the database and its JDBC driver must implement XA
- JTA (Java Transaction API)
  - consists of javax.transaction packages
  - builds on X/Open DTP
  - defines the contracts between application, app server, transaction manager and resource manager
  - defines UserTransaction class to be used by J2EE developer

Transaction fundamentals 4/4

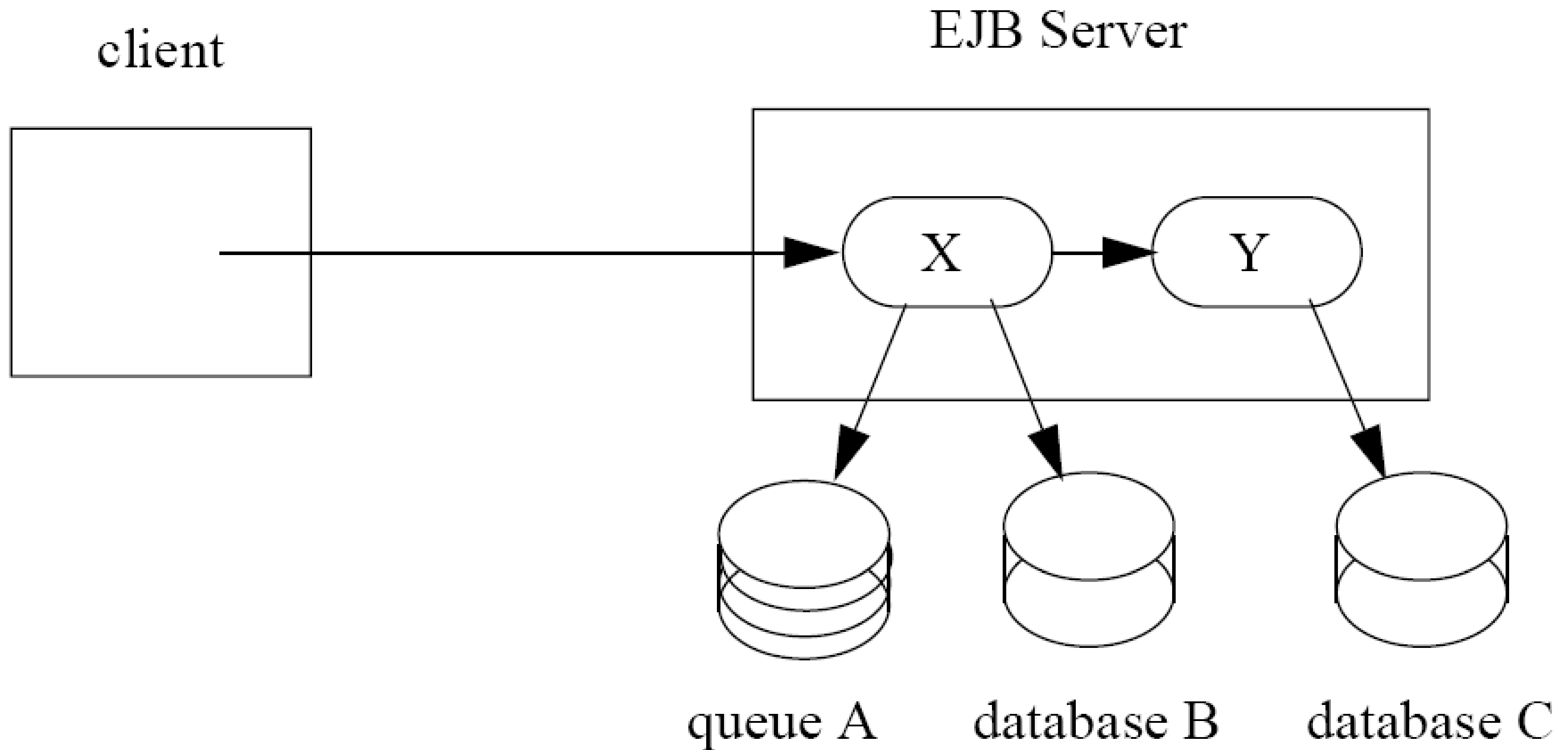
- Two-phase commit is the protocol executed by the transaction manager in a distributed transaction to ensure ACID properties
  - e.g. in a successful scenario:



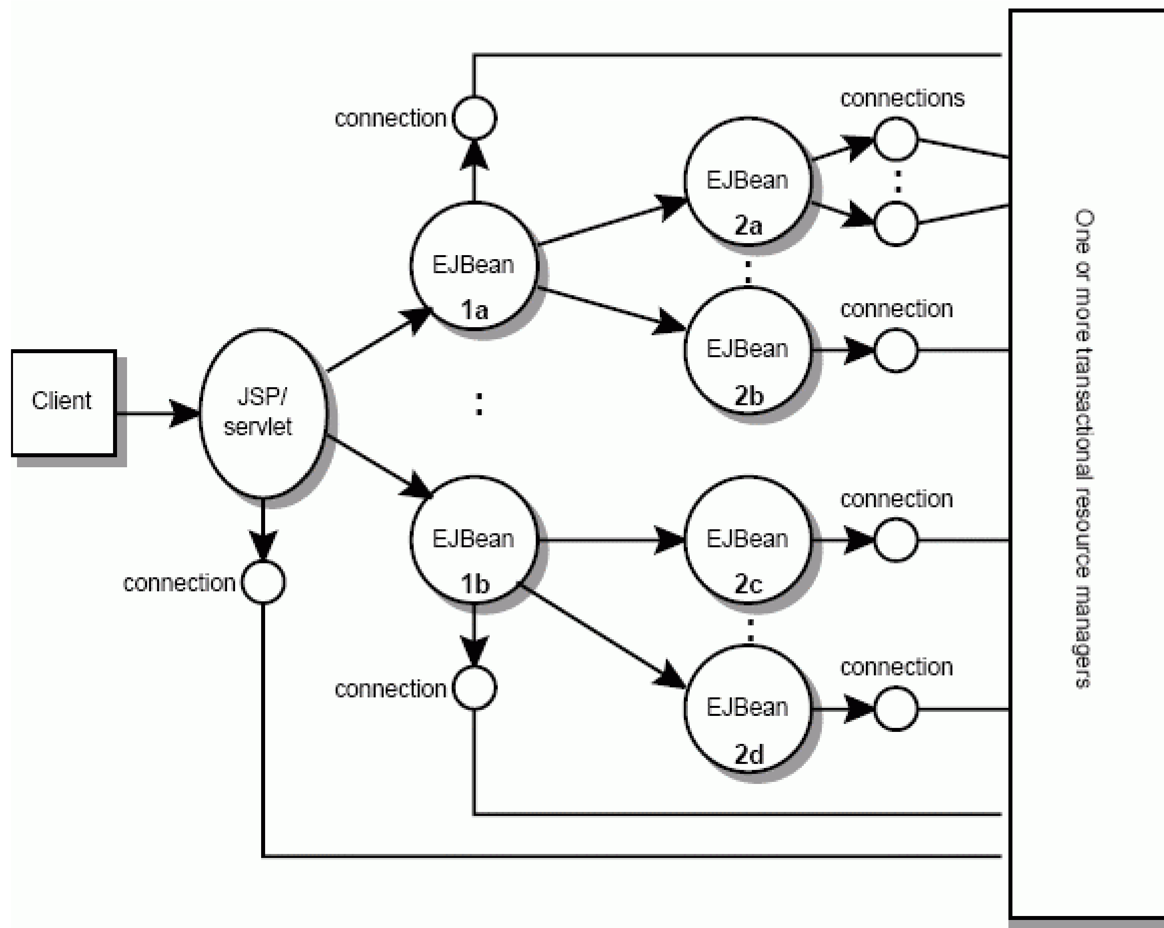
Transactions in J2EE

- Distributed transactions must be supported by app server
  - involve multiple transactional resources
  - involve multiple components (in particular EJBs)
  - app server contains transaction manager that coordinates two-phase commit across multiple XA-capable resources
- Transactional resources in J2EE:
  - RDBMS accessed via JDBC connection
  - MOM (message-oriented middleware) accessed via JMS session
  - EIS accessed via resource adapter (connector)
  - Some resources (or their adapters) may not support XA!
- IIOP transaction propagation protocol currently not required (= transaction managers need not be implemented in terms of JTS)
- Only flat transactions (no nested transactions)

A distributed transaction scenario



Another distributed transaction scenario



Transaction attributes for container-managed tx

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NotSupported	none	none	none
	T1	none	none
Required	none	T2	T2
	T1	T1	T1
Supports	none	none	none
	T1	T1	T1
RequiresNew	none	T2	T2
	T1	T2	T2
Mandatory	none	**error**	N/A
	T1	T1	T1
Never	none	none	none
	T1	**error**	N/A

Exceptions thrown from EJB methods 1/3

- Application exceptions
  - are all checked exceptions (not a sub-class of RuntimeException) that are not sub-classes of RemoteException
  - are used to report business logic problems, not technical problems
    - the intention of an application exception is to signal to the client that the EJB ran into an expected condition that prevents it from fulfilling the request, e.g. illegal arguments supplied to the EJB method, precondition of calling the EJB method not met
  - e.g. javax.ejb.CreateException, javax.ejb.RemoveException, com.sun.tasktracker.....EntryServiceArgumentException
  - can be thrown from any method in home or component interface
- An application exception thrown from an EJB method does not cause an automatic rollback of a pending transaction: in the EJB method you need to
  - either explicitly rollback transaction (EJBContext.setRollbackOnly())
  - or make sure that a commit leads to a consistent state of the EJB

Exceptions thrown from EJB methods 2/3

- System exceptions
  - are all sub-classes of RuntimeException including EJBException
    - EJB methods must not throw RemoteException
  - are used to report unexpected problems or problems that the EJB can not recover from
    - e.g., OutOfMemoryError occured in method body, inability to obtain database connection/make JNDI lookup/send JMS message, unexpected RuntimeExceptions occured in method body
  - should be thrown by the EJB method as follows:
    - EJB method bodies should not catch RuntimeException (but pass it through as a system exception)
    - EJB method bodies must catch unrecoverable checked exceptions (e.g. JNDI's NamingException) and throw an EJBException
    - throw EJBException if anything else unrecoverable happens

Exceptions thrown from EJB methods 3/3

- A system exception thrown from an EJB method is caught by the EJB container and
  - causes a rollback of the current transaction (regardless of whether the transaction was started by the container or by the bean (and is not commited or rolled back))
  - causes the client to receive a RemoteException (for remote clients) or EJBException (for local clients)
  - the EJB instance is not used by the container any more
- The EJB method does not have to worry about clean-up if it throws a system exception
    - transction is rolled-back (see before)
    - the container releases any resources (DB connections, etc.) that are declared in the EJB's environment (!)

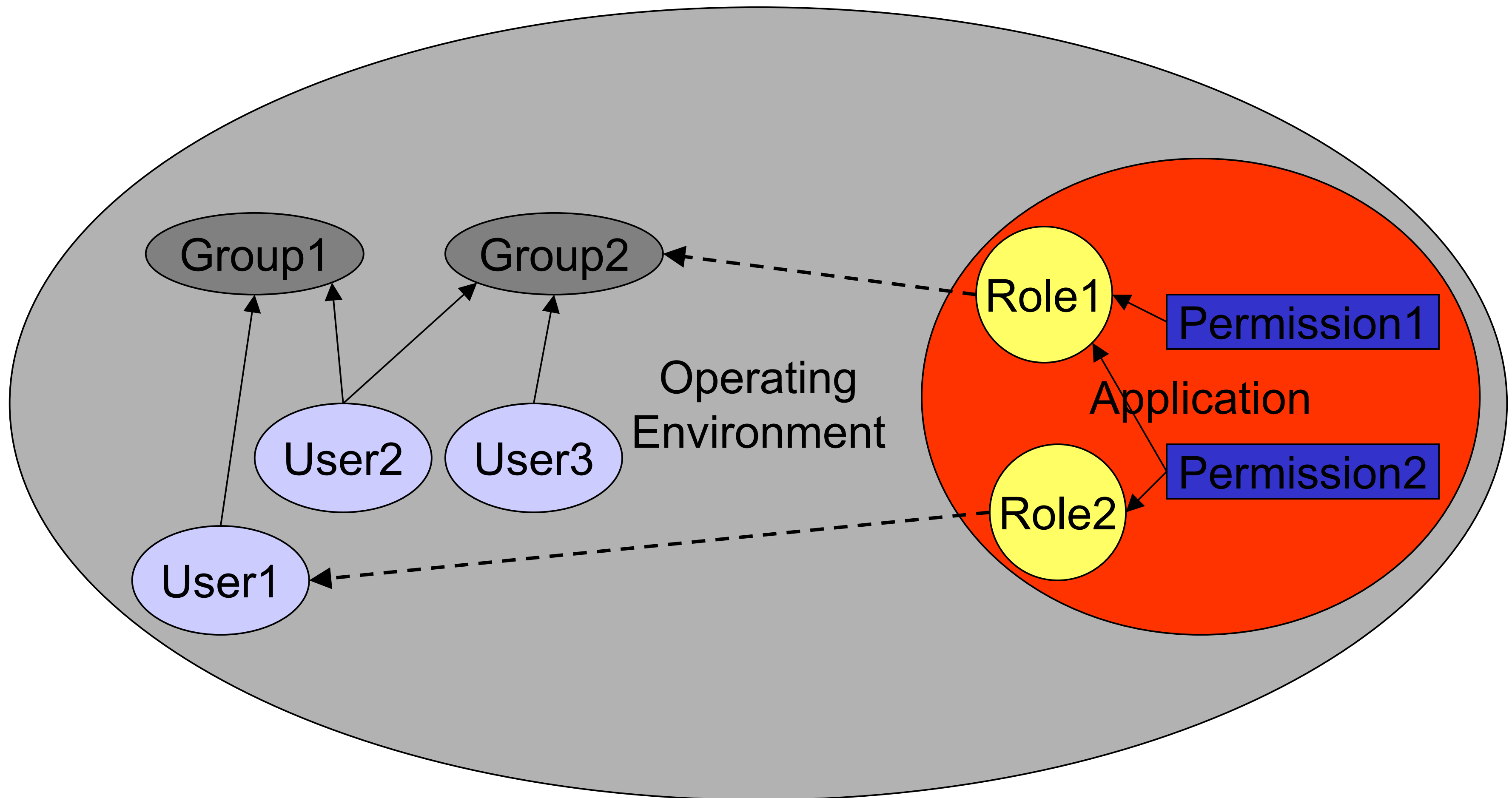
Security terminology

- Principal
  - The authenticated subject, e.g. the user
  - Has a name
- Authentication data
  - E.g. the password
- Authentication
  - The process of proving the identity of a principal
  - Involves matching principal and authentication data against a "store" (LDAP, database, file)
    - Username and password are validated
- Credential
  - Encodes what the user is allowed to do
  - Is the result of successful authentication

Security terminology

- (Security) role
  - A logical concepts that is used by the application and Java EE to group permissions
  - An application defines security roles, which need to be mapped to principals (users) from the operating environment
  - E.g. admin, manager, loser
- Authorization, access control
  - The process of granting of denying a principal access to resources based on the principal's roles
- Security (policy) domain, realm
  - Scope of one security policy
- Securiy context
  - Used by app server to hold credentials

Security terminology 3/3



Know what you do when using a remote view

- Applies to remote client view and web service client view
- Martin Fowler's first law of distributed objects: "Don't distribute your objects"
- Location independence is beautiful and provides flexibility in deployment but remote calls
  - have high latency (network, network stack, marhsalling/unmarshalling ("copying") of parameters and return values)
  - must therefore be coarse-grained: few remote calls, transporting as much data as is sensible and possible
  - may fail due to network problems, unavailable server, etc.
- Local EJBs offer mainly "only" (declarative) security and transaction support over normal Java objects
- The developer decides between remote and/or local client view

Passivation and activation

- Applies to stateful session beans and entity beans
- The app server (ejb container) actively manages memory by serializing/deserializing bean instances to/from disk when required: passivation/activation
  - all fields in an EJB must be serializable, "part of the EJB spec" (or null at the moment of passivation)
  - don't use transient fields
- ejbPassivate() and ejbActivate() methods called by container immediately before passivation and after activation, respectively
  - ejbPassivate()
    - close any open resources (e.g. DB connections)
    - set all fields to null that are not serializable or "part of the EJB spec"
  - ejbActivate()
    - re-open any resources (e.g. DB connections)
    - re-initialize any null-fields

Declarative security for EJBs

- Protectable resource: calling EJB methods
- An application's roles and access controls are declared in the deployment descriptor
  - again: roles are a logical concept of the application; all roles must be enumerated in the deployment descriptor
  - permissions (to call EJB methods) are assigned to roles
  - roles are mapped to principals (users) and groups from the operating environment (e.g. in the app server specific deployment descriptor)
- Example: Task Tracker ejb-jar.xml, sun-application.xml

Programmatic security for EJBs

- Encoding authorisation requirements in code
- To be used (only) if declarative security is not enough (too static)
- API
  - EJBContext
    - isCallerInRole(String roleName)
      - can be used to code more dynamic security policies than can be expressed (in the deployment descriptor) using declarative security
    - getCallerPrincipal()
      - could be used to lookup information in database based on the name of the principal calling the EJB
- Role names used in code are logical role names that can be linked to "real" role names in the deployment descriptor (security-role-ref).
- Example: Task Tracker EntryServiceBean.createEntry()

What we didn't talk about

- Enterprise bean context and lookup
  - Explicit lookups:
    - New `EJBContext.lookup()` method
    - JNDI lookups
- New deployment descriptor

Implementing web services

HTTP GET request

```
GET /articles/news/today.asp HTTP/1.1
Accept: */*
Accept-Language: en-us
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/links.asp
User-Agent: Mozilla/4.0 (compatible; MSIE
  5.5; Windows NT 5.0)
Accept-Encoding: gzip, deflate
```

HTTP response

```
HTTP/1.1 200 OK
Date: Wed, 13 Jan 1999 13:19:42 GMT
Server: Apache/1.3.1 (Unix)
Connection: close
Cache-control: private
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
  4.0Transitional//EN">
<HTML>
…
</HTML>
```

Need to know

- Web service concepts
  - HTTP
  - SOAP
  - WSDL
  - (UDDI)
- WSDL-first versus Java-first
- Essential JAX-WS 2.0 and JAXB 2.0 annotations/concepts
  - see tasktrackerWSExample
- Keep this in mind:
  - Document-centric rather than RPC
  - Literal rather than encoded
  - Wrapped rather than unwrapped

JavaServer Faces

HTTP basics

- RFC 1945 (HTTP/1.0), RFC 2616 (HTTP/1.1)
  - http://www.rfc.net/
- Request -Response cycle
- HTTP Request
  - Method: GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE
  - Request URL
  - Header, body
- HTTP Response
  - Result code: 404 (not available), 500 (server error)
  - Header, body

HTML forms – HTTP GET

```
<form action="http://localhost/serv"
  method="GET">
 name=<input type="text" name="name">
 age=<input type="text" name="age">
 <input type="submit" VALUE="get this!">
</form>



http://localhost/serv?name=anton&age=35
```

HTML forms – HTTP POST

```
<form action="http://localhost/serv"
  method="POST">
 name=<input type="text" name="name">
 age=<input type="text" name="age">
 <input type="submit" VALUE="post this!">
</form>



http://localhost/serv
```

Servlet request URL

- Available from HttpServletRequest
- http://www.you.com/superapp/buy/confirm?value=OK
  - Protocol: http
  - Host: www.you.com
  - Request path: /superapp/buy/confirm
  - Context path: /superapp
  - Servlet path: /buy
  - Path info: /confirm
  - Query string: value=OK

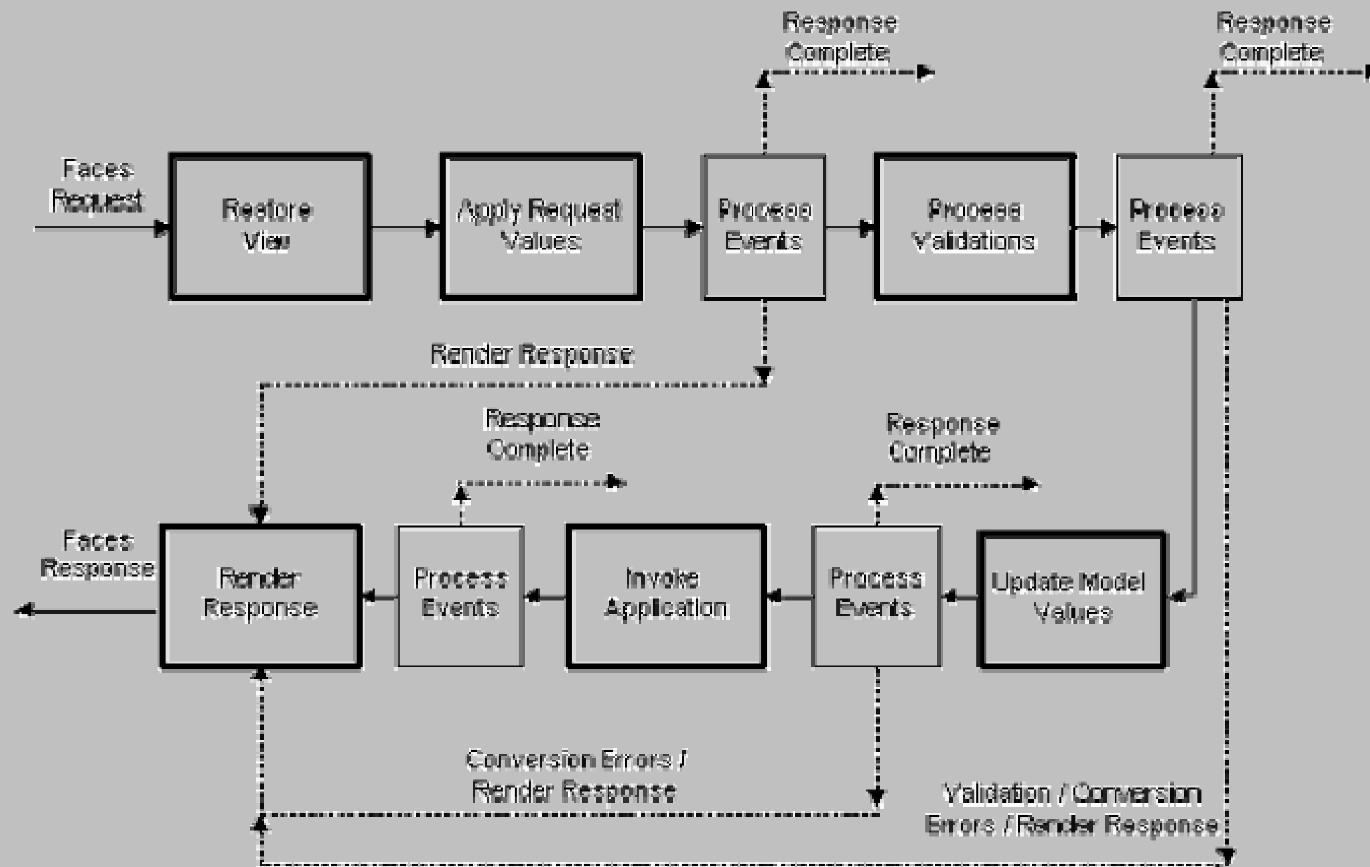
Servlet (web) sessions

- Session ties together HTTP requests from one client
- HttpServletRequest.getSession() of all requests (within a session) returns the same session object
  - call before getting Writer/OutputStream so that a cookie can be set
- Session identity maintained between requests
  - via cookies (stored on client and sent with requests)
  - via URL rewriting (http://host/a/b/c.jsp;jsessionid=12321): must be done explicitly through HttpServlerResponse.encodeURL()
- Sessions timeout and/or call to invalidate()
- Objects can be stored in session as named attributes
  - HttpSession.setAttribute(String, Object) and getAttribute(String)
  - all attributes should be serilizable!

Need to know

- Using JSPs as the view technology
- Managed beans and their scope
- Calling SBs in the service layer from managed beans
- Configuring the controller: faces-config.xml

JSF request processing lifecycle



JSF request processing life cycle

- 3 representative calls from JSF to model
  - Validation
    - `<h:inputText … validator="#{model.valid}"/>` calls `void valid(FacesContext, UIComponent, Object)` on "model"
  - Get/set model values
    - `<h:inputText value="#{model.size}"/>` calls setter/getter for "size" on "model"
  - Invoke application ("action")
    - <h:commandButton action="#{mode.doit}"/> calls `String doit()` on "model"
- Prerequisites: ELResolver, value/method expressions

As an aside: ELResolver

- New in 1.2 for VariableResolver and PropertyResolver
- Resolves segments in an expression
  - #{model.size} is resolved to a JavaBean called "model" and its property "size"
- There is a ManagedBeanELResolver that finds JSF managed beans by name
- There can also be a SessionBeanResolver that does a JNDI-lookup of EJB 3 session beans…
  - JBoss Seam does this
  - (Spring similarly resolves Spring beans)

Managed Beans and Java EE 5

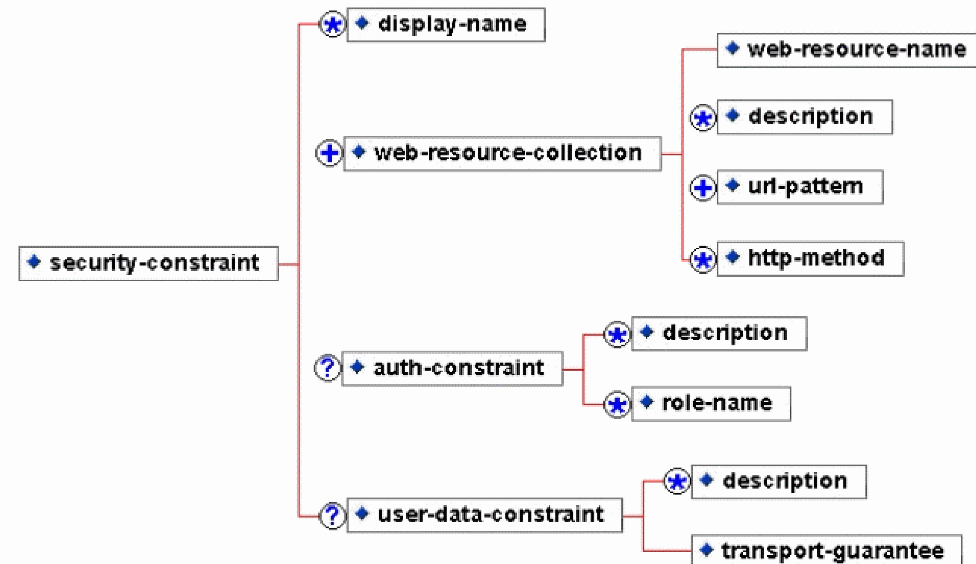
- JavaBeans created/destroyed by JSF's Managed Bean facility are known as "Managed Beans"
- Managed Beans may be the target of Java EE 5 dependency injection:
  - `@EJB`, `@Resource`
  - `@PersistenceContext` not yet?

Declarative security for web components 1/2

- An application's roles and access controls are declared in the deployment descriptor
- Protectable resource: URLs
- Web authentication mechanisms:
  - HTTP basic authentication
    - username/password (base64-encoded)
    - handled by web browser
  - HTTP form-based authentication
    - username/password, but app provides HTML form
  - HTTPS client authentication
    - user needs Public Key Certificate

Declarative security for web components 2/2

- Transport guarantee is the means of specifying encrypted communication (i.e. HTTPS)
- Security roles are identified in the deployment descriptor (web.xml) and mapped to users/groups from the operating environment in the app-server-specific deployment descriptor (sun-web.xml or sun-application.xml)
- Example: TaskTracker web.xml, sun-web.xml, sun-application.xml, admin console



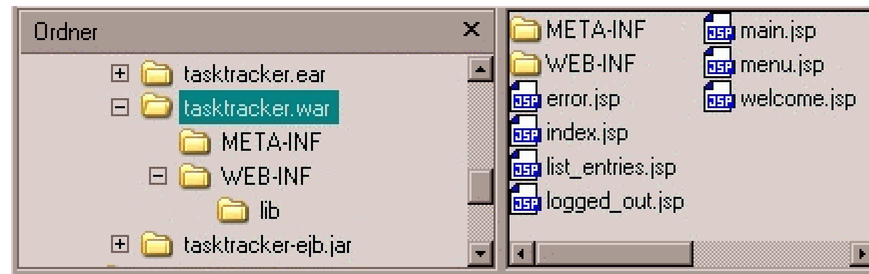
Programmatic security for web components

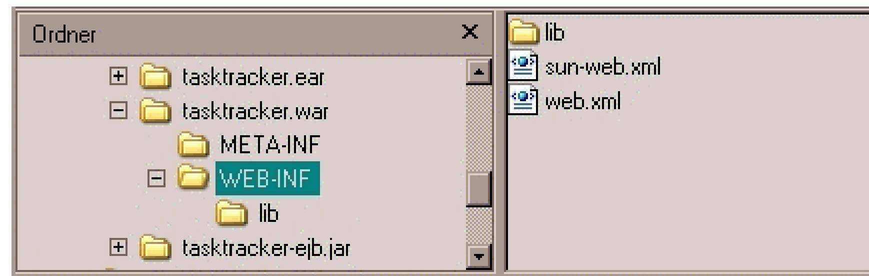
- Encoding authorisation requirements in code
- To be used (only) if declarative security is not enough (too static)
- API
  - HttpServletRequest
    - getRemoteUser(): the login name of the user making the request (if (!) sent with the request)
    - isUserInRole(String roleName)
    - getUserPrincipal(): the name of the currently authenticated user (wrapped in a Principal object); null means not logged in
- Role names used in code are logical role names that can be linked to "real" role names in the deployment descriptor (security-role-ref).

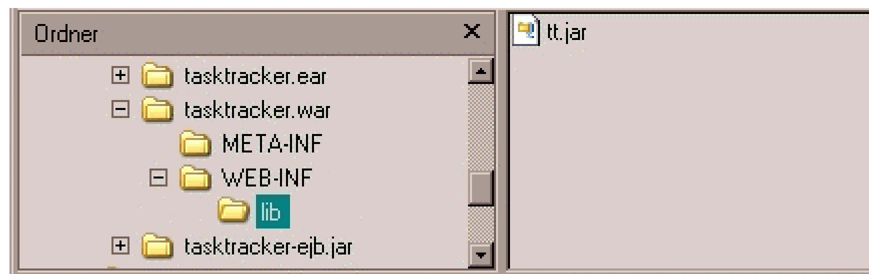
Web archive (WAR) 1/2

- Packaging and deployment unit of web apps
- Contains
  - web components (servlets, JSPs)
  - server-side Java classes
  - static web content (HTML, images, …)
  - client-side Java classes (applets, support classes)
  - standard and app-server-specific deployment descriptor (web.xml and sun-web.xml)
- Packaged as jar with extension war
  - jar cvf example.war .

Web archive (WAR) 2/2





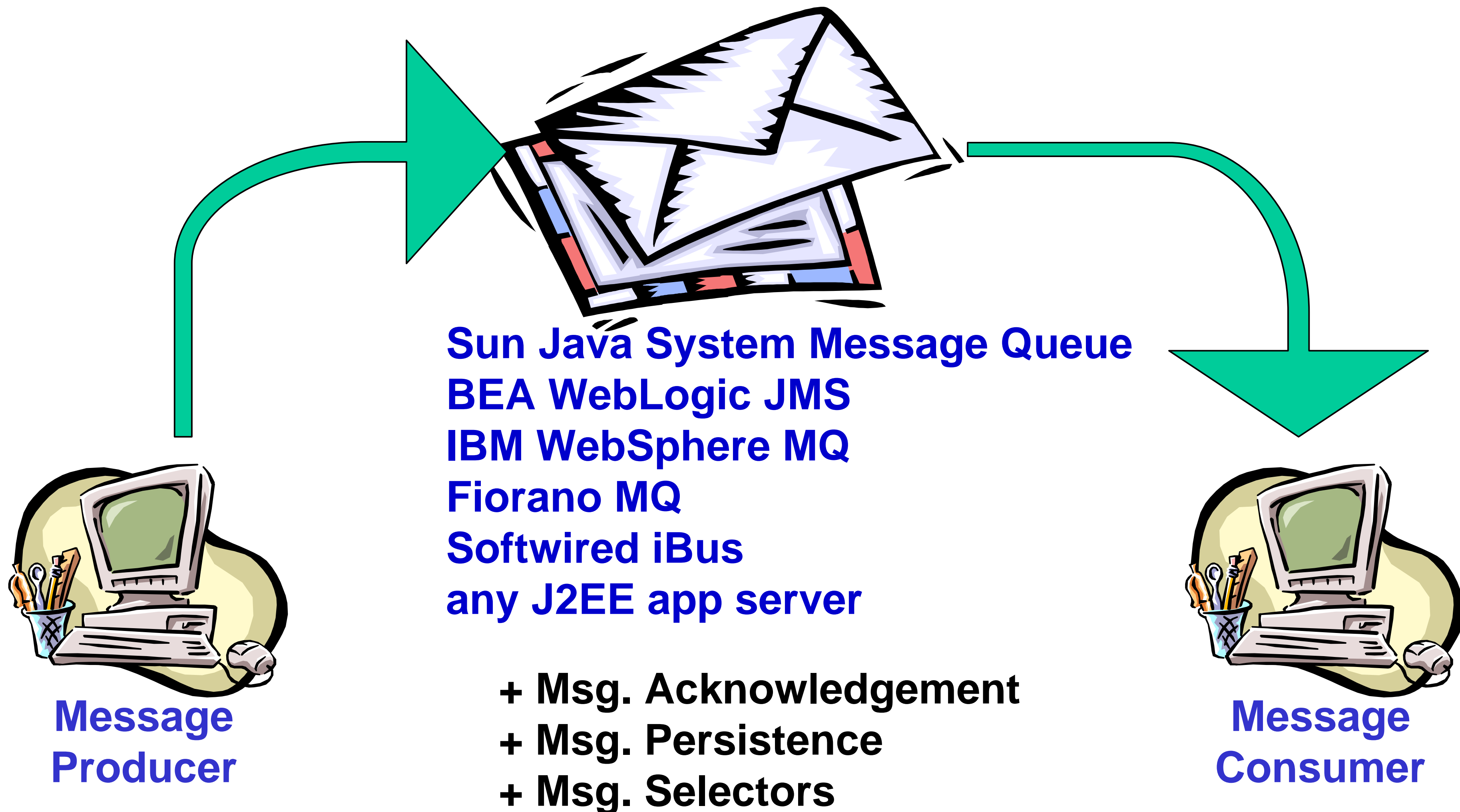


Asynchronous server-side Java

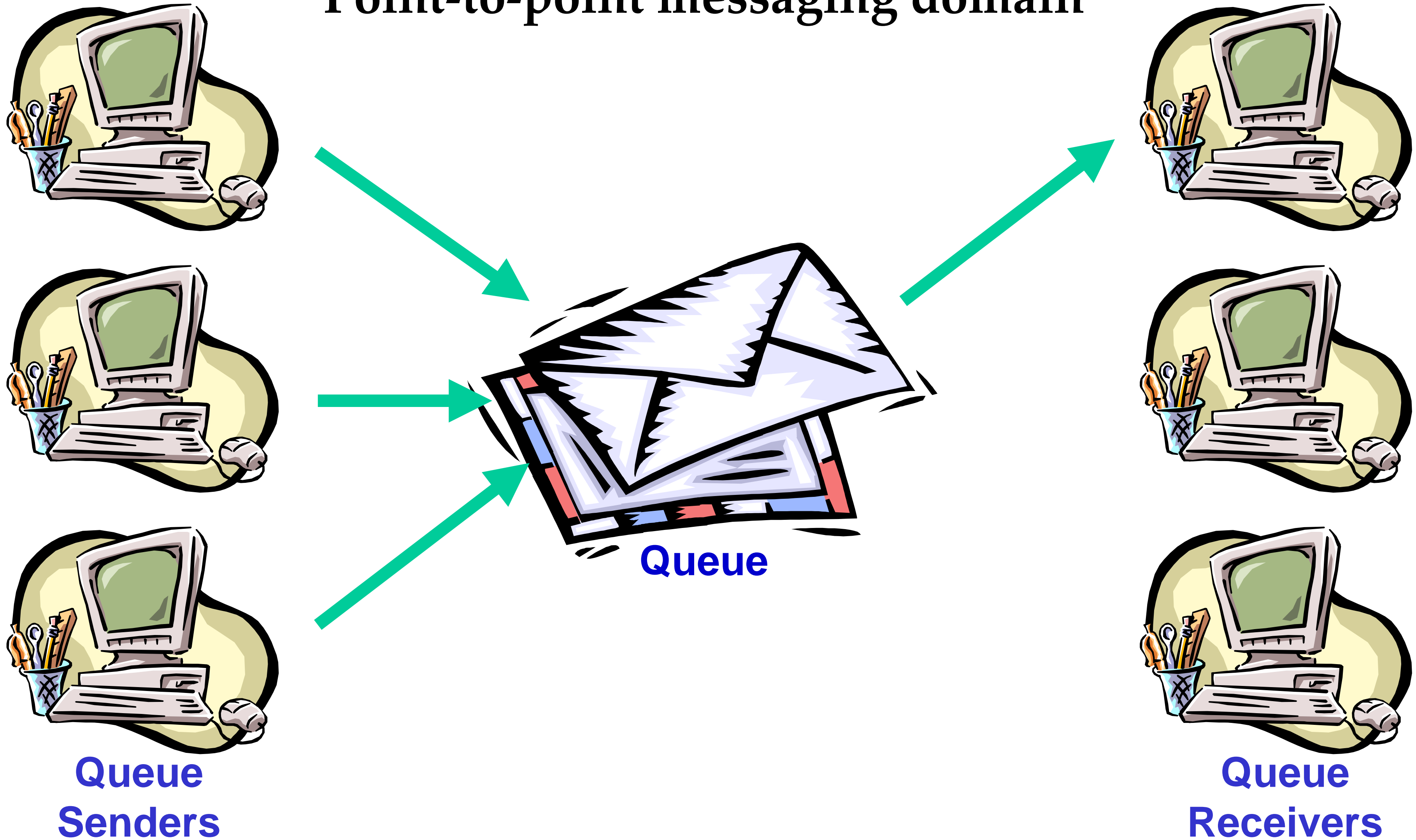
MOM, JMS and message-driven beans overview

- Message-oriented middleware (MOM) is a class of enterprise software products that facilitates the programmatic exchange of messages
  - this is not email
  - messaging is peer-to-peer via MOM
  - possesses typical enterprise features: reliability, transaction support, scalability, security, ...
- Java Messaging Service (JMS) is the Java API to MOM
  - MOM-product is called JMS provider
  - supports queues and topics
- Every J2EE app server contains a JMS provider (MOM product)
- A message-driven bean (MDB) is an EJB that consumes messages via JMS

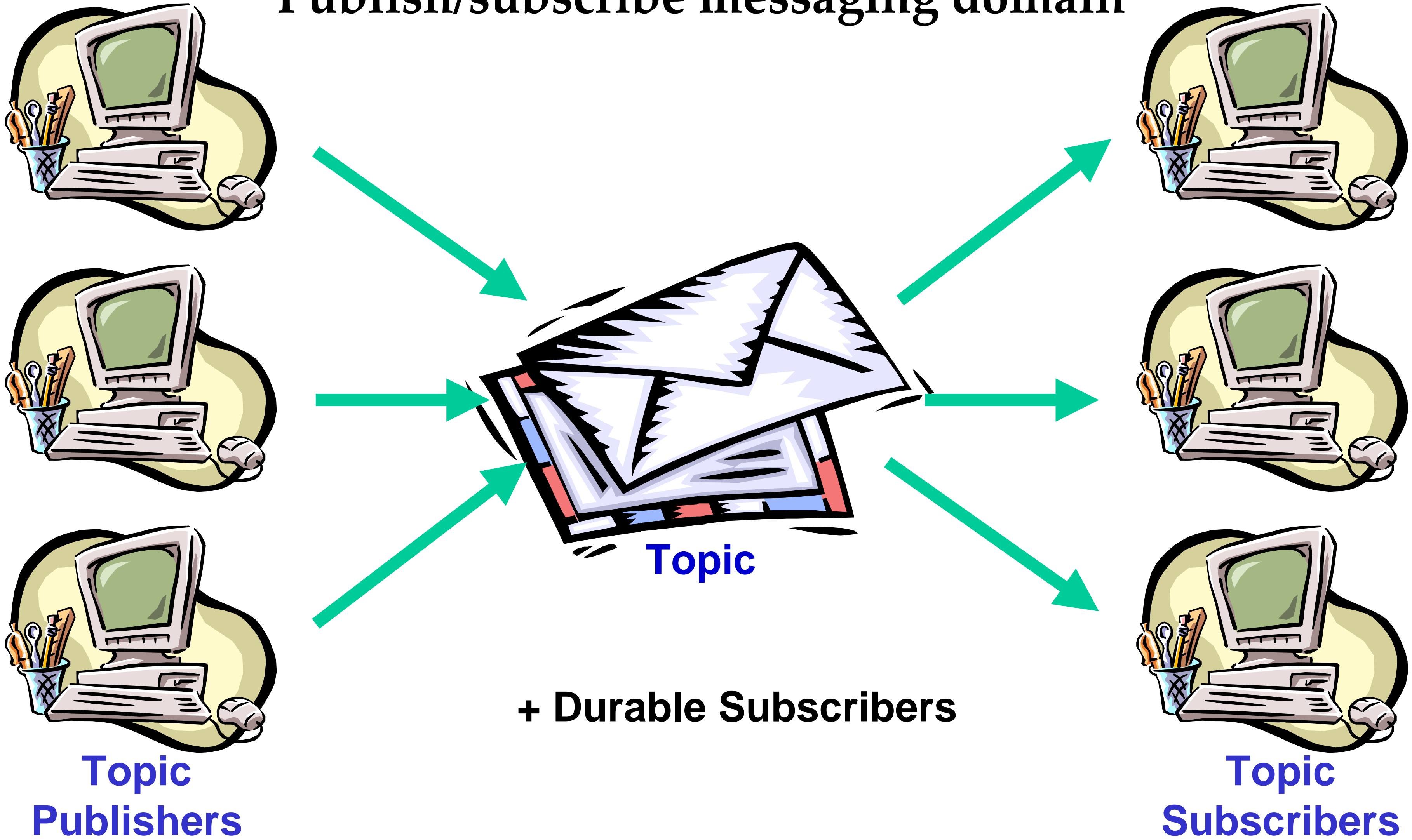
Messaging



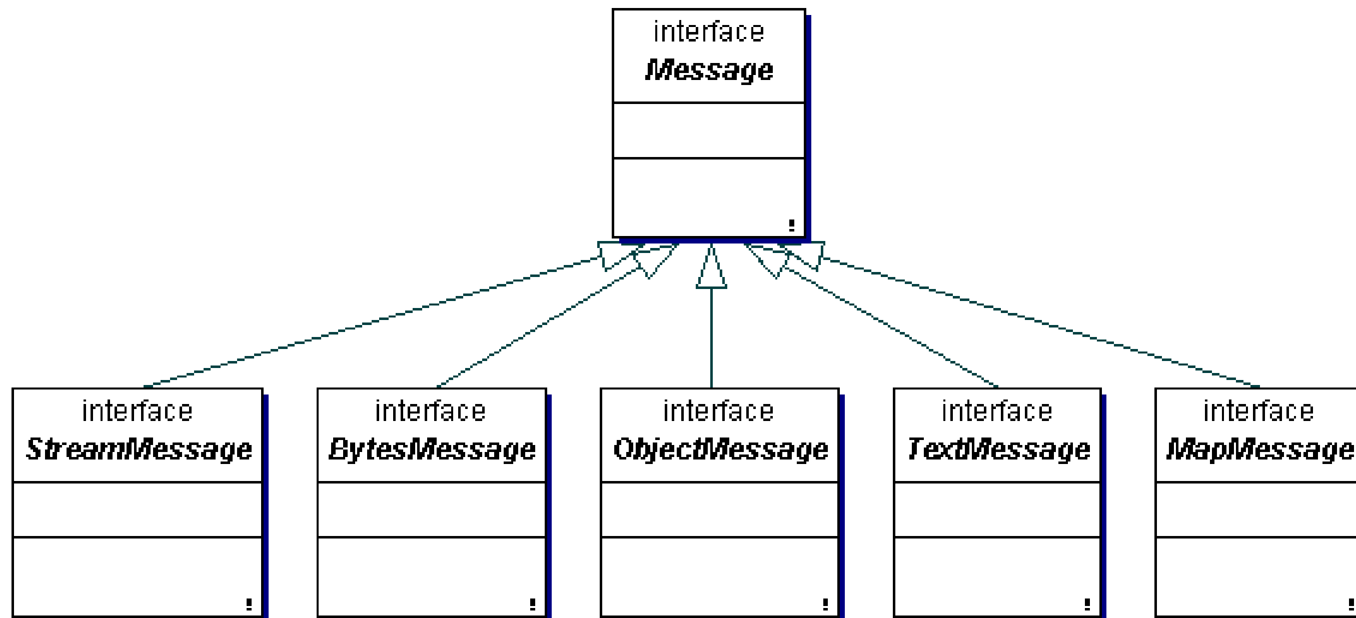
Point-to-point messaging domain



Publish/subscribe messaging domain



Message types



+ Properties
+ Format Conversion

JMS message producer

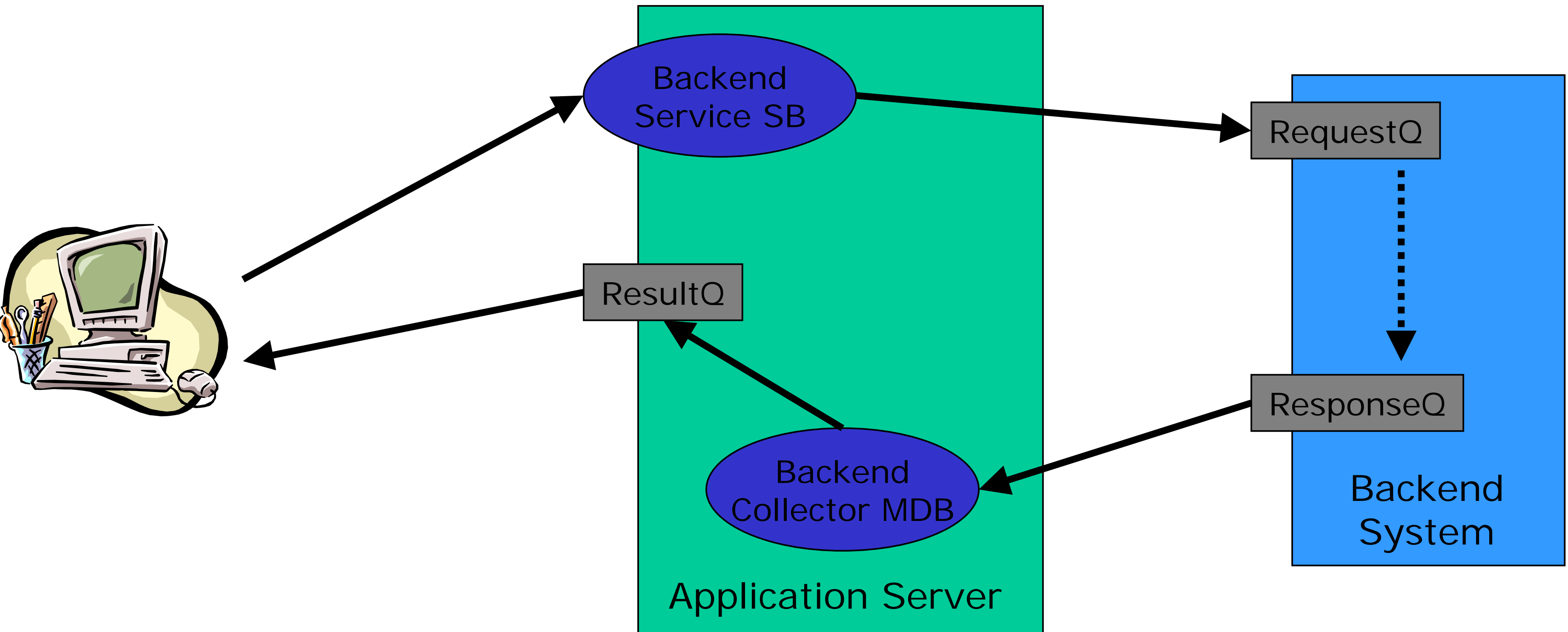
```
public class HelloQueueSender {
    public static final String D_NAME  = "ex1Queue";
    public static final String CF_NAME = "QueueConnectionFactory";

    public static void main(String[] args) {
        try {
            Context                ctx   = new InitialContext();
            QueueConnectionFactory qcf   = (QueueConnectionFactory) ctx.lookup(CF_NAME);
            Queue                  q     = (Queue) ctx.lookup(D_NAME);
            QueueConnection        qc    = qcf.createQueueConnection();
            try {
                QueueSession       qsess = qc.createQueueSession(false,
                                                Session.AUTO_ACKNOWLEDGE);
                QueueSender        qsnd  = qsess.createSender(q);
                TextMessage        msg   = qsess.createTextMessage("Hello JMS World");
                qsnd.send(msg);
            } finally {
                try {qc.close();} catch (Exception e) {}
            }
        } catch (Exception e) {
            System.out.println("Exception occurred: " + e.toString());
        }
    }
}
```

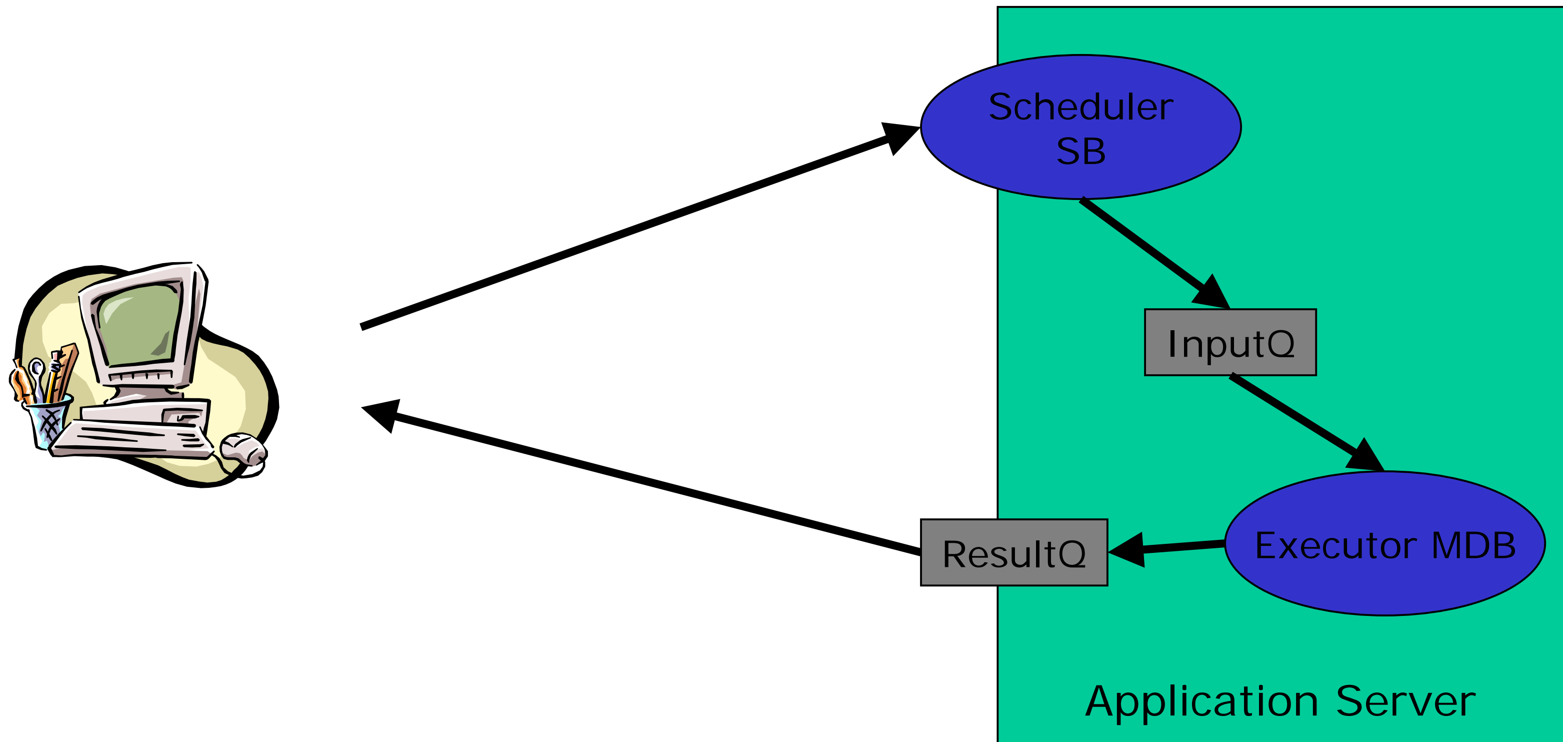
Message-driven bean characteristics

- Asynchronous message consumer from queues and/or topics
- MDB can also listen on non-JMS messages
- Stateless
- No home interface or component interface
  - an MDB implements the interface MessageListener which defines one method onMessage(Message)
- Client does not interact directly with MDB
  - client sends message to queue/topic
  - JMS provider delivers message to MDB
  - complete decoupling of client and MDB
    - client does not know of existence of MDB
    - MDB does not know client identity (principal, caller, user, whatever)
- MDBs are a high-performance EJB type that is known only on the app server

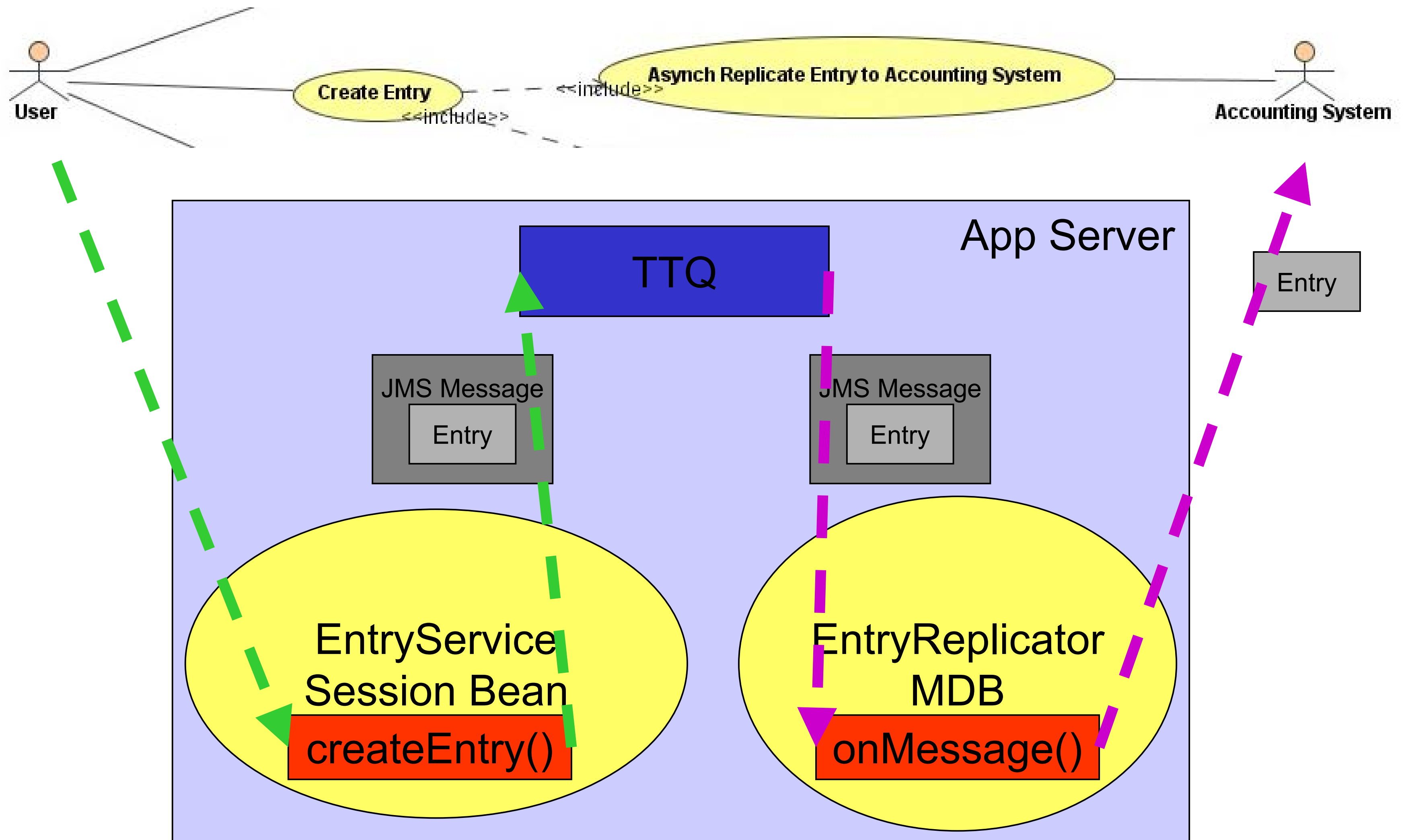
Messaging scenario 1: asynch backend connectivity



Messaging scenario 2: job scheduling



Task Tracker and messaging



Message-driven beans

- `@MessageDriven` annotation
  - No need to implement `MessageDrivenBean`
- Business interface is defined by messaging type
  - For JMS: `javax.jms.MessageListener`
- Lifecycle callbacks
  - `@PostConstruct`, `@PreDestroy`

AsynchNappyChanger.java

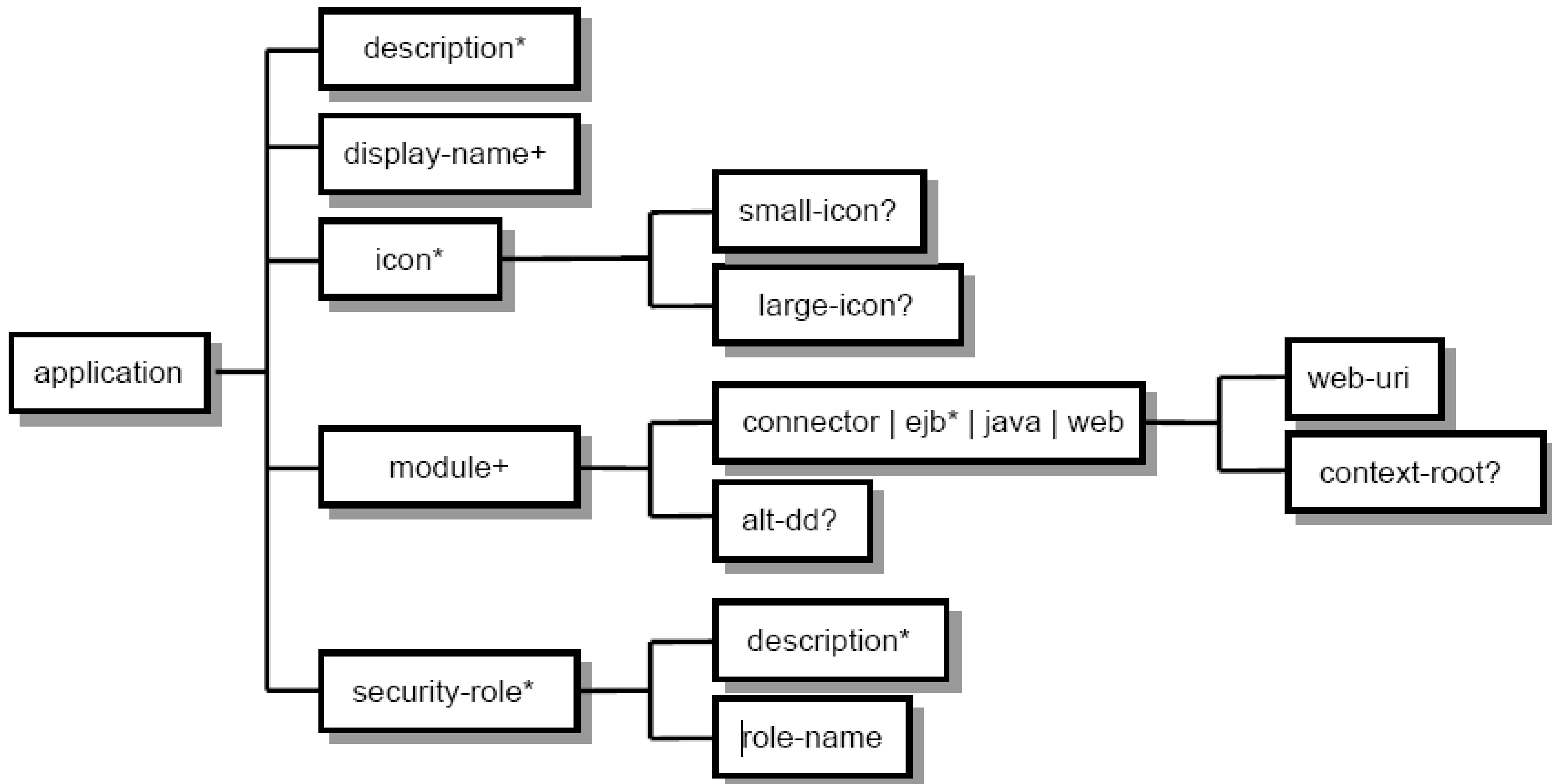
```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName="destinationType",
    propertyValue="javax.jms.Queue"),
  @ActivationConfigProperty(
    propertyName="destination",
    propertyValue="jms/NappyQ")
})
public class AsynchNappyChanger implements
    MessageListener {
  public void onMessage(Message msg) {
    // ...
  }
}
```

J2EE application review

- A J2EE application bundles several modules into one ear-file
- Deployment descriptors (application.xml, sun-application.xml) describe the application as a whole and certain properties of each module
  - if you know that modules will be used within an application, then move DD elements into the application DDs to avoid redundancy:
    - context-root for web modules
    - security-role-mapping from role names to users/groups
- Example: Task Tracker application.xml, sun-application.xml

J2EE application (ear) deployment descriptor

- META-INF/application.xml in .ear file



Summary: Need to know

Need to know

- Core concepts
  - layering
  - Java EE 5 technology mapping to layers
- Java Persistence API (EJB 3 persistence)
  - Entity, Embeddable
  - ID, persistence context, EntityManager
    - persist() vs. merge()
    - transcaction-scoped persistence context
  - Basic mappings: primitives, String, Date, etc.
  - Relationship mapping:
    - one-to-many, many-to-one
    - (one-to-one, many-to-many)
  - Lazy vs. eager loading, problem with detached objects

Need to know

  - Entity states and detached object use
  - Cascading of entity state changes
- EJB 3 Simplified API
  - Session bean types: stateful, stateless
  - Message driven beans
  - Container managed security (declarative security)
  - Bean-managed security (programmatic security)
  - Container managed transactions
    - Transaction attributes
  - Transaction propagation
- JSF web applications
  - Managed beans
  - JSPs as views

Need to know

  - Essential JSF HTML components
  - Simplified JSF request processing cycle
  - Authentication (login-config)
  - Authorization (security constraint, declarative security)
  - Programmatic security
- Security context propagation
  - web to EJB
  - EJB to EJB
- Implementing web services with JAX-WS 2.0
  - anatomy of a WSDL definitions file
  - Simple Java-first approach
  - Essential JAXB 2.0 annotations to guide Java-XML mapping: field access, transient

Need to know

- JMS and MDBs
  - Message types
  - Destination types: Queue and Topic
  - Transactional MDBs and message retrieval
  - Transactional SBs and message sending

thank you !

dr. gerald loeffler
enterprise software architect, shipserv ltd
gerald.loeffler@googlemail.com