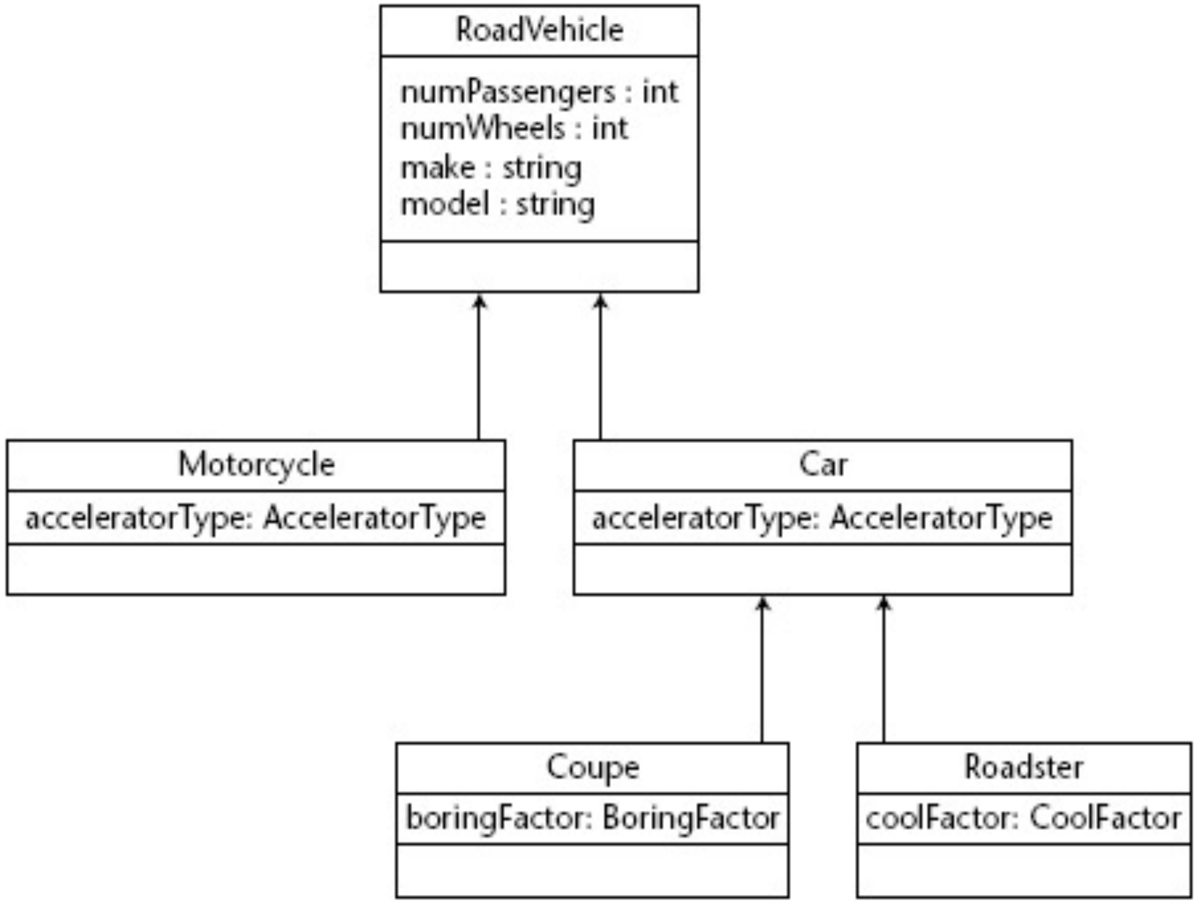
# Advanced Persistence

Inheritance

# Mapping inheritance



# SINGLE TABLE PER CLASS

| Id | numPass | numWheels | make | model |
|---|---|---|---|---|
| 1 | 6 | 2 | HORSE CART | NULL |

| Id | numPass | numWheels | make | model | acceleratorType |
|---|---|---|---|---|---|
| 2 | 1 | 2 | HONDA | HRC7 | THROTTLE |

etc.

**Problems with polymorphism – how do you find "all RoadVehicles that have less than 3 passenger?"**

# SINGLE TABLE PER CLASS HIERARCHY

| Id | numPass | numWheels | make | model | DISC | acceleratortype | BoringFactor | CoolFactor |
|---|---|---|---|---|---|---|---|---|
| 1 | 6 | 2 | HORSECART | NULL | ROADVEHICLE | NULL | NULL | NULL |
| 2 | 1 | 2 | HONDA | HRC7 | MOTORCYCLE | THROTTLE | NULL | NULL |
| 3 | 4 | 4 | FIAT | PUNTO | CAR | PEDAL | NULL | NULL |
| 4 | 2 | 4 | FERRARI | F70 | COUPE | PEDAL | 1 | NULL |
| 5 | 2 | 4 | FORD | KA | ROADSTER | PEDAL | NULL | 1 |

- **Space inefficiency**
- **Impossible to set “NON-NULL” constraints on fields of the subclasses.**

# JOINED TABLES

RoadVehicle

| Id | DTYPE | numPass | numWheels | make | model |
|---|---|---|---|---|---|
| 1 | ROADVEHICLE | 6 | 2 | HORSECART | NULL |
| 2 | MOTORCYCLE | 1 | 2 | HONDA | HRC7 |
| 3 | CAR | 4 | 4 | FIAT | PUNTO |
| 4 | COUPE | 2 | 4 | FERRARI | F70 |
| 5 | ROADSTER | 2 | 4 | FORD | KA |

Car

| Id | acceleratortype |
|---|---|
| 3 | PEDAL |
| 4 | PEDAL |
| 5 | PEDAL |

Coupe

| Id | boringFactor |
|---|---|
| 4 | 1 |

**Many joins in a deep inheritance hierarchy – time inefficiency.**

# The base class

```
package examples.entity.single_table;
// imports go here
@Entity(name="RoadVehicleSingle")
@Table(name="ROADVEHICLE") //optional, it's the default
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISC",
   discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("ROADVEHICLE")
// @Inheritance(strategy=InheritanceType.JOINED)
public class RoadVehicle implements Serializable {
    public enum AcceleratorType {PEDAL,THROTTLE};
    @Id
    protected int id;
    protected int numPassengers;
    protected int numWheels;
    protected String make;
    protected String model;
    public RoadVehicle() {
        id = (int) System.nanoTime();
    }
    // setters and getters go here
    ...
}
```

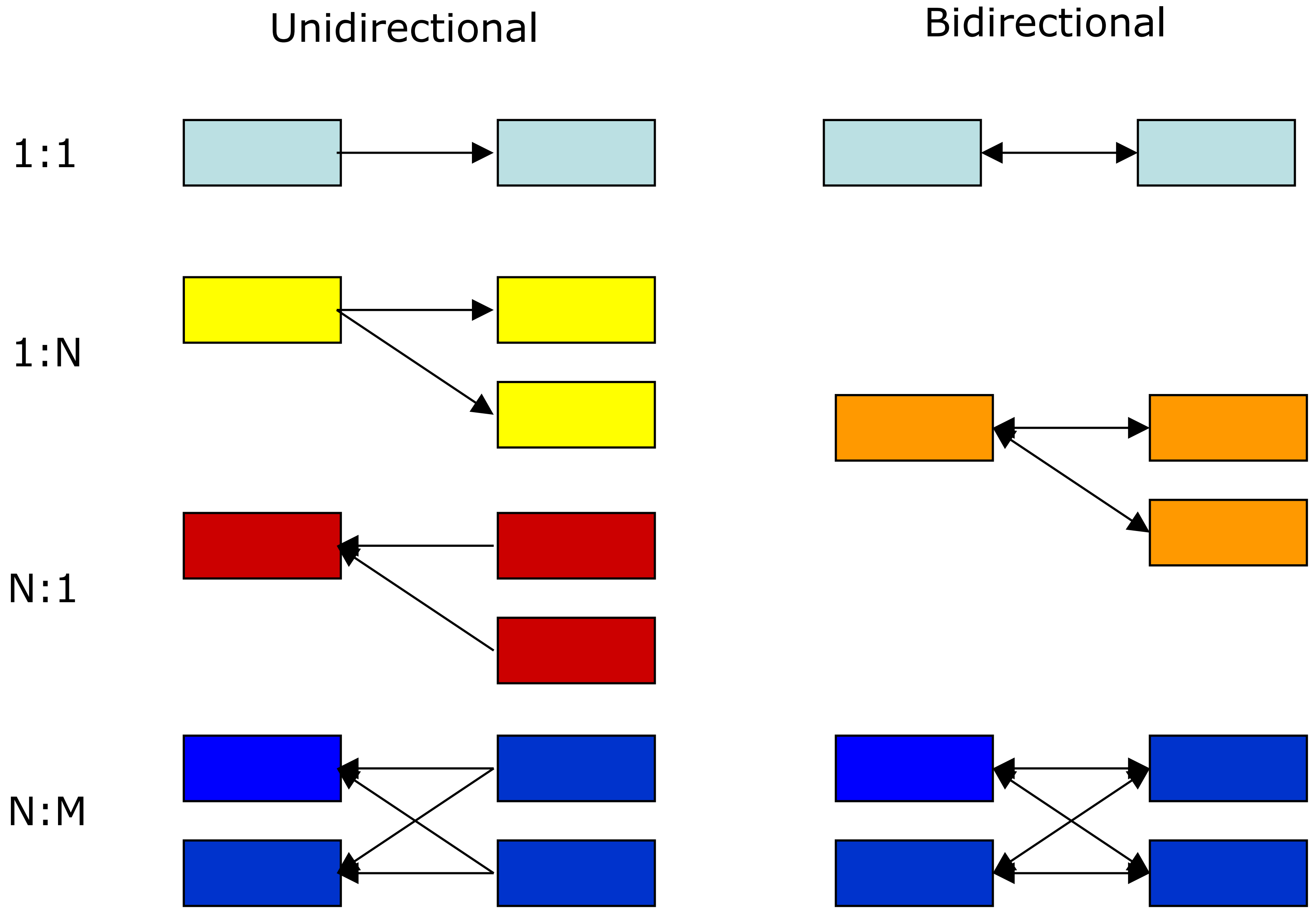
# The derived class

```
package examples.entity.single_table;
// imports go here
@Entity
@DiscriminatorValue("MOTORCYCLE") //not needed for joined
public class Motorcycle extends RoadVehicle implements
   Serializable {
    public final AcceleratorType acceleratorType
       =AcceleratorType.THROTTLE;
    public Motorcycle() {
        super();
        numWheels = 2;
        numPassengers = 2;
    }
}
```

# Advanced Persistence

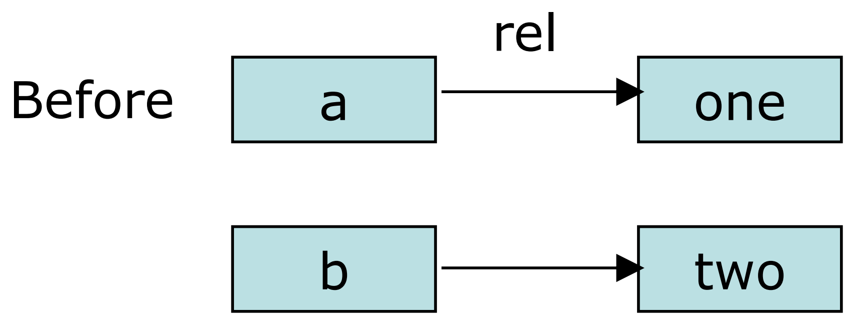
## Relationships

# Multiplicity and Directionality – 7 types

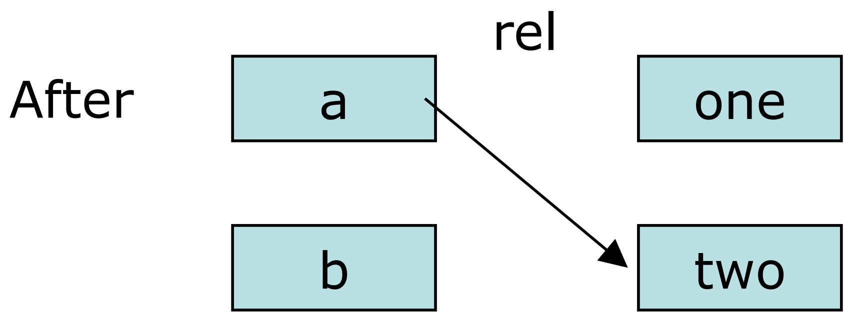


# Watch out for side effects!

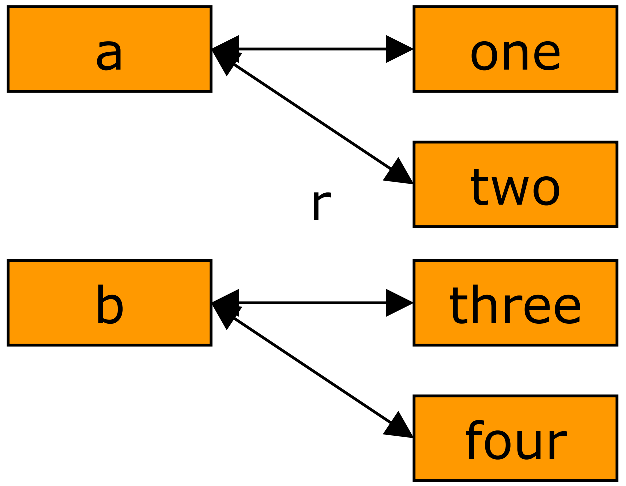
Let rel be a 1:1 relationship



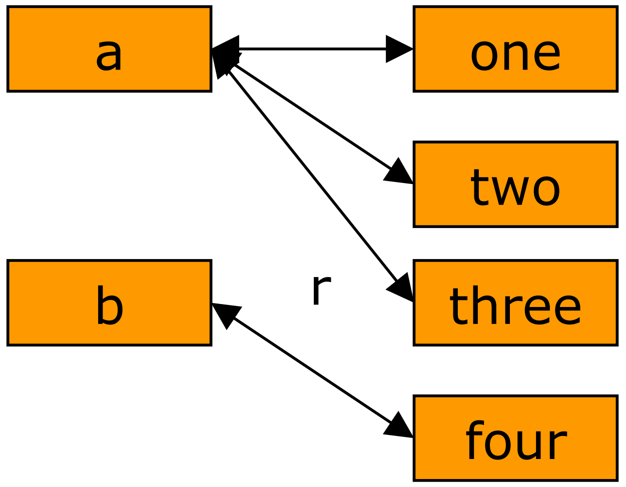
a.setRel(two)



Let r be a 1:N relationship

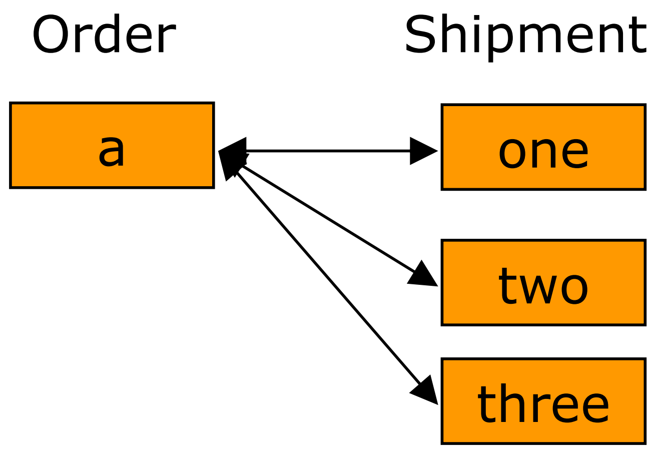


a.setR(three)



# Cascade-delete

When we delete “a”,
should also one,two e three
be canceled?



# Relation – 1:1 unidir – “from”

```
@Entity(name=”OrderUni”)
public class Order implements Serializable {
    private int id;
    private String orderName;
    private Shipment shipment;
    public Order() { id = (int)System.nanoTime(); }
    @Id
    public int getId() { return id; }
    public void setId(int id) {
    this.id = id;
    }
    ...
    // other setters and getters go here
    ...
    @OneToOne(cascade={CascadeType.PERSIST})
    public Shipment getShipment() {
        return shipment;
    }
    public void setShipment(Shipment shipment) {
        this.shipment = shipment;
    }
}
```

## Relation – 1:1 unidir – “to”

```
...
@Entity(name=”ShipmentUni”)
public class Shipment implements Serializable {
    private int id;
    private String city;
    private String zipcode;
    public Shipment() { id = (int)System.nanoTime(); }
    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    ...
    // other setters and getters go here
}
```

# Relation – 1:1 unidir – client

```
...
@Stateless
public class OrderShipmentUniBean implements OrderShipment {
    @PersistenceContext
    EntityManager em;
    public void doSomeStuff() {
        Shipment s = new Shipment();
        s.setCity("Austin");
        s.setZipcode("78727");
        Order o = new Order();
        o.setOrderName("Software Order");
        o.setShipment(s);
        em.persist(o);
    }
    public List getOrders() {
        Query q = em.createQuery("SELECT o FROM OrderUni o");
        return q.getResultList();
    }
}
```

## Relation – 1:1 bidir – “to”

```
...
@Entity(name=”ShipmentUni”)
public class Shipment implements Serializable {
    private int id;
    private String city;
    private String zipcode;
    private Order order;
    public Shipment() { id = (int)System.nanoTime(); }
    @Id
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    ...
    // other setters and getters go here
    ...
    @OneToOne(mappedBy=”shipment”)
    // shipmentproperty from the Order entity
    public Order getOrder() {
        return order;
    }
    public void setOrder(Order order) {
        this.order = order;
    }
}
```

# Relation – 1:1 bidir – client

```
...
@Stateless
public class OrderShipmentUniBean implements OrderShipment {
    @PersistenceContext
    EntityManager em;
    public void doSomeStuff() {
        Shipment s = new Shipment();
        s.setCity("Austin");
        s.setZipcode("78727");
        Order o = new Order();
        o.setOrderName("Software Order");
        o.setShipment(s);
        em.persist(o);
    }
    public List getOrders() {
        Query q = em.createQuery("SELECT o FROM OrderUni o");
        return q.getResultList();
    }
    ..
    public List getShipments() {
        Query q = em.createQuery("SELECT s FROM Shipment s");
        return q.getResultList();
    }
}
```

# Relation – 1:N unidir – “from”

```
...
@Entity(name=”CompanyOMUni”)
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;
    ...
    // other getters and setters go here
    // including the Id
    ...
    @OneToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER)
        public Collection<Employee> getEmployees() {
        return employees;
    }
    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
}
```

## Relation – 1:N unidir – “to”

```
...
@Entity(name=”EmployeeOMUni”)
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;
    ...
    // other getters and setters go here
    // including the Id
    ...
}
```

# Relation – 1:N unidir – client

```
Company c = new Company();
c.setName(“M*Power Internet Services, Inc.”);Collection<Employee>
   employees = new ArrayList<Employee>();
Employee e = new Employee();
e.setName(“Micah Silverman”); e.setSex(‘M’); employees.add(e);
e = new Employee();
e.setName(“Tes Silverman”); e.setSex(‘F’); employees.add(e);
c.setEmployees(employees);
em.persist(c);
c = new Company();
c.setName(“Sun Microsystems”);
employees = new ArrayList<Employee>();
e = new Employee();
e.setName(“Rima Patel”); e.setSex(‘F’); employees.add(e);
e = new Employee();
e.setName(“James Gosling”); e.setSex(‘M’); employees.add(e);
c.setEmployees(employees);
em.persist(c);
```

# Relation – 1:N bidir – “from”

```
...
@Entity(name=”CompanyOMUni”)
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;
    ...
    // other getters and setters go here
    // including the Id
    ...
    @OneToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER,
    mappedBy=”company”)
        public Collection<Employee> getEmployees() {
        return employees;
    }
    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
}
```

## Relation – 1:N bidir – “to”

```
...
@Entity(name=”EmployeeOMUni”)
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;
    private Company company;
    ...
    // other getters and setters go here
    // including the Id
    @ManyToOne
    public Company getCompany() {
      return company;
    }
    public void setCompany(Company company) {
      this.company = company;
    }
}
```

# Relation – M:N

**The rules for generating a join table are:**

**1. The name of the join table will be the name of the owning entity, followed by an underscore (_), followed by the name of the target entity.**

**2. The name of the first column in the join table will be the property name, followed by an underscore, followed by the primary key name in the owner entity.**

**3. The name of the second column in the join table will be the property name, followed by an underscore, followed by the primary key name in the target entity.**

**4. The types of the columns in the join table will match the primary key types of the tables that will be referenced by it.**

# Relation – M:N unidir – “from”

```
...
@Entity(name="StudentUni")
public class Student implements Serializable {
    private int id;
    private String name;
    private Collection<Course> courses = new ArrayList<Course>();
    public Student() { id = (int)System.nanoTime(); }
    @Id
    public int getId() { return id; }
    ...
    //other setters and getters go here
    ...
    @ManyToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER)
    @JoinTable(name="STUDENTUNI_COURSEUNI")
    public Collection<Course> getCourses() {
        return courses;
    }
    public void setCourses(Collection<Course> courses) {
        this.courses = courses;
    }
}
```

## Relation – M:N unidir – “to”

```
...
@Entity(name=”CourseUni”)
public class Course implements Serializable {
private int id;
private String courseName;
private Collection<Student> students = new ArrayList<Student>();
...
//setters and getters go here
...
}
```

# Relation – M:N bidir – “from”

```
...
@Entity(name=”StudentUni”)
public class Student implements Serializable {
    private int id;
    private String name;
    private Collection<Course> courses = new ArrayList<Course>();
    public Student() { id = (int)System.nanoTime(); }
    @Id
    public int getId() { return id; }
    ...
    //other setters and getters go here
    ...
    @ManyToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER)
    @JoinTable(name=”STUDENTUNI_COURSEUNI”)
    public Collection<Course> getCourses() {
        return courses;
    }
    public void setCourses(Collection<Course> courses) {
        this.courses = courses;
    }
}
```

# Relation – M:N bidir – “to”

```
...
@Entity(name=”CourseBid”)
public class Course implements Serializable {
private int id;
private String courseName;
private Collection<Student> students = new ArrayList<Student>();
    ...
    //getters and setters go here
    ...
    @ManyToMany(cascade={CascadeType.ALL},
    fetch=FetchType.EAGER,mappedBy=”courses”)
        public Collection<Student> getStudents() {
        return students;
    }
    public void setStudents(Collection<Student> students) {
        this.students = students;
    }
}
```

## Transactions in EJB

## Declarative Transactions

## Who begins a transaction?

*Who* begins a transaction? *Who* issues either a commit or abort? This is called *demarcating transactional boundaries* .

There are three ways to demarcate transactions:

•*programmatically:*

*you* are responsible for issuing a *begin* statement and either a *commit* or an *abort* statement.

•*declaratively,*

the EJB container *intercepts* the request and starts up a transaction automatically on behalf of your bean.

•*client-initiated.*

write code to start and end the transaction from the client code outside of your bean.

## Programmatic vs. declarative

*programmatic transactions:*

***your bean has full control over transactional boundaries.****For instance,you can use programmatic transactions to run a series of minitransactions within a bean method.*

*When using programmatic transactions,always try to complete your transactions in the same method that you began them.Doing otherwise results in spaghetti code where it is difficult to track the transactions;the performance decreases because the transaction is held open longer.*

*declarative transactions:*

***your entire bean method must either run under a transaction or not run under a transaction.***

*Transactions are simpler! (just declare them in the descriptor)*

## Client-initiated

***Client initiated transactions:***

*A non-transactional remote client calls an enterprise bean that performs its own transactions The bean succeeds in the transaction,but the network or application server crashes before the result is returned to a remote client.The remote client would receive a Java RMI RemoteException indicating a network error,but would not know whether the transaction that took place in the enterprise bean was a success or a failure.*

***With client-controlled transactions, if anything goes wrong,the client will know about it.***

*The downside to client-controlled transactions is that if the client is located far from the server, **transactions are likely to take a longer time and the efficiency will suffer**.*

## Transactional Models

A *flat transaction* is the simplest transactional model to understand.A flat transaction is a series of operations that are performed atomically as a single *unit of work* .

A *nested transaction* allows you to embed atomic units of work within other units of work.The unit of work that is nested within another unit of work can roll back without forcing the entire transaction to roll back. (*subtransactions can independently roll back without affecting higher transactions in the tree*)
(Not currently mandated by the EJB specification)

Other models: *chained transactions* and *sagas*.
(Not supported by the EJB specification)

## EJB Transaction Attribute Values

***Required***

***You want your method to always run in a transaction.***

*If a transaction is already running,your bean joins in on that transaction. If no transaction is running,the EJB container starts one for you.*

***Never***

***Your bean cannot be involved in a transaction.***

*If the client calls your bean in a transaction,the container throws an exception back to the client (java.rmi.RemoteException if remote, javax.ejb.EJBException if local).*

## EJB Transaction Attribute Values

***Supports***

***The method runs only in a transaction if the client had one running already*** *—it joins that transaction.*
*If the client does not have a transaction,the bean runs with no transaction at all.*

*Mandatory*

***a transaction must be already running when your bean method is called****. If a transaction isn 't running, javax.ejb.TransactionRequiredException is thrown back to the caller (or javax.ejb.TransactionRequiredLocalException if the client is local).*

## EJB Transaction Attribute Values

***NotSupported***

***your bean cannot be involved in a transaction at all.***

*For example,assume we have two enterprise beans,A and B.Let ′s assume bean A begins a transaction and then calls bean B. If bean B is using the* NotSupported *attribute,the transaction that A started is suspended. None of B′s operations are transactional,such as reads/writes to databases. When B completes,A ′s transaction is resumed.*

EJB Transaction Attribute Values

***RequiresNew***

*You should use the RequiresNew attribute if you **always want a new transaction to begin** when your bean is called. If a transaction is already underway when your bean is called,that transaction is suspended during the bean invocation.*

*The container then launches a new transaction and delegates the call to the bean.The bean performs its operations and eventually completes.The container then commits or aborts the transaction and finally resumes the old transaction. If no transaction is running when your bean is called,there is nothing to suspend or resume.*

## EJB Transaction Attribute Values

| TYPE | PRECONDITION | POSTCONDITION |
|---|---|---|
| **Required** | NO transaction | NEW |
| | PRE-EXISTING | PRE-EXISTING |
| **RequiresNew** | NO transaction | NEW |
| | PRE-ESISTENTE | NEW |
| **Supports** | NO transaction | NO transaction |
| | PRE-EXISTING | PRE-EXISTING |
| **Mandatory** | NO transaction | error |
| | PRE-EXISTING | PRE-EXISTING |
| **NotSupported** | Nessuna transazione | NO transaction |
| | PRE-ESISTENTE | NO transaction |
| **Never** | NO transaction | NO transaction |
| | PRE-EXISTING | error |

# Annotations

- @Stateless
- @TransactionManagement(javax.ejb.TransactonManagementType.CONTAINER)
- public class Mybean implements Myinterface{
- @PersistencyContext private EntityManager em;
- @Resource private SessionContext ctx;
- …
- @TransactionAttribute(javax.ejb.TransactonManagementType.REQUIRED)
-         public void myTransactedMethod(){…
-                 if (…) ctx.setRollbackOnly;
-         }
- }

## EJB Transaction Attribute Values

```
<assembly-descriptor>
    <container-transaction>
       <method>
             <ejb-name>Employee</ejb-name>
             <method-name>*</method-name>
       </method>
       <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
    <container-transaction>
       <method>
             <ejb-name>Employee</ejb-name>
             <method-name>setName</method-name>
             <method-param>String</method-param>
       </method>
       <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

## Dooming container-managed transactions

**call *setRollbackOnly()* on your *EJB context* object.**
**If the transaction participant is not an Container Managed EJB component, you can doom a transaction by looking up the JTA and calling the JTA 's *setRollbackOnly()* method,**

**Container-managed transactional beans can detect doomed transactions by calling the *getRollbackOnly()*method on the EJB context object. If this method returns *true* ,the transaction is doomed.**

## Isolation levels in EJB

***BMT:***

***you specify isolation levels with your resource manager API (such as JDBC).***

*For example,you could call java.sql.Connection.SetTransactionIsolation(...).*

***CMT:***

***there is no way to specify isolation levels in the deployment descriptor.***

*You need to either use resource manager APIs (such as JDBC),or rely on your container 's tools or database 's tools to specify isolation.*

## Isolation portability problems

**Unfortunately, there is no way to specify isolation for container-managed transactional beans in a portable way—you are reliant on container and database tools.**

**This means if you have written an application, you cannot ship that application with built-in isolation. The deployer now needs to know about transaction isolation when he uses the container's tools, and the deployer might not know a whole lot about your application's transactional behavior.**