Java Persistence with Hibernate Fundamentals

INTRODUCING ORM AND ITS PROBLEMS

Cătălin Tudose
PHD IN COMPUTER SCIENCE, JAVA AND WEB TECHNOLOGIES EXPERT
www.catalintudose.com
https://www.linkedin.com/in/catalin-tudose-847667a1

Overview

ORM (Object-relational mapping) and JPA (Java Persistence API)

Advantages and drawbacks of Hibernate

Object-relational impedance mismatch

Simple Hibernate application

What Is ORM?

Object-relational mapping

Storing the representation of the objects

JPA and Hibernate

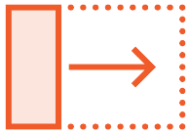
Advantages of JPA and Hibernate

- Write less code
- Quicker development
- Exempt from knowing SQL
- Consistent model to interact with the database
- Independent of the database vendor

Drawbacks of JPA and Hibernate

- Learning curve
- Harder to debug
- Performance may suffer
- JDBC is closer to the database
- Use specific features of a vendor database

Object-relational Impedance Mismatch



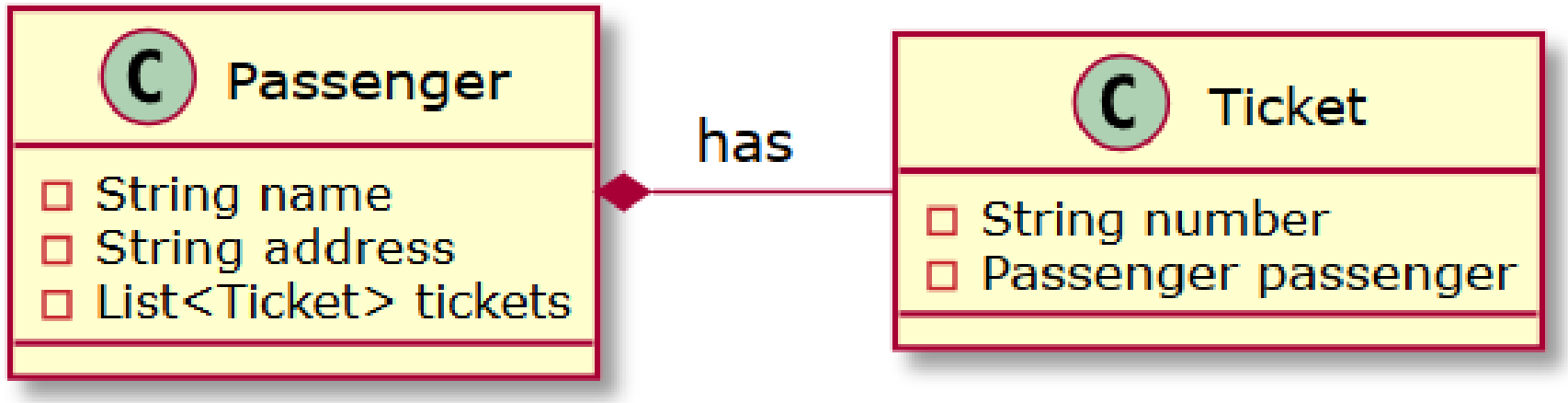
Object and relational models do not work fine together

Interconnected objects vs. related tables

Granularity, inheritance, identity, associations, and data navigation

The Granularity Problem

The Flights Management Application



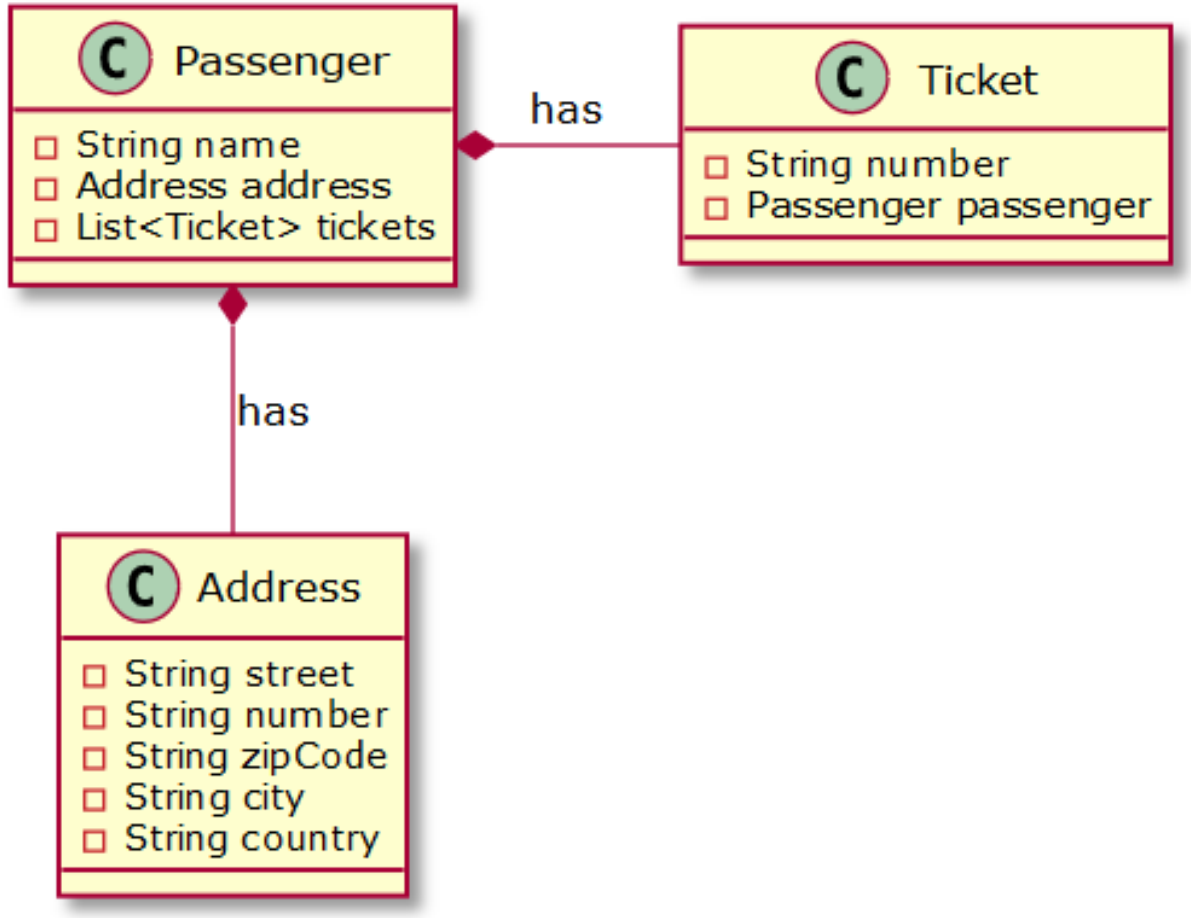
The Flights Management Classes

```
public class Passenger {
    private String name;
    private String address;
    private List<Ticket> tickets;
}

public class Ticket {
    private String number;
    private Passenger passenger;
}
```

The Flights Management Tables

```
create table PASSENGERS (
        NAME varchar(255),
        ADDRESS varchar(255),
        primary key (NAME)
)

create table TICKETS (
        NUMBER varchar(255),
        PASSENGER_NAME varchar(255),
        primary key (NUMBER)
)

alter table TICKETS
       add constraint FK_PASSENGERS
       foreign key (PASSENGER_NAME)
       references PASSENGERS (NAME)
```

The Extended Flights Management Application

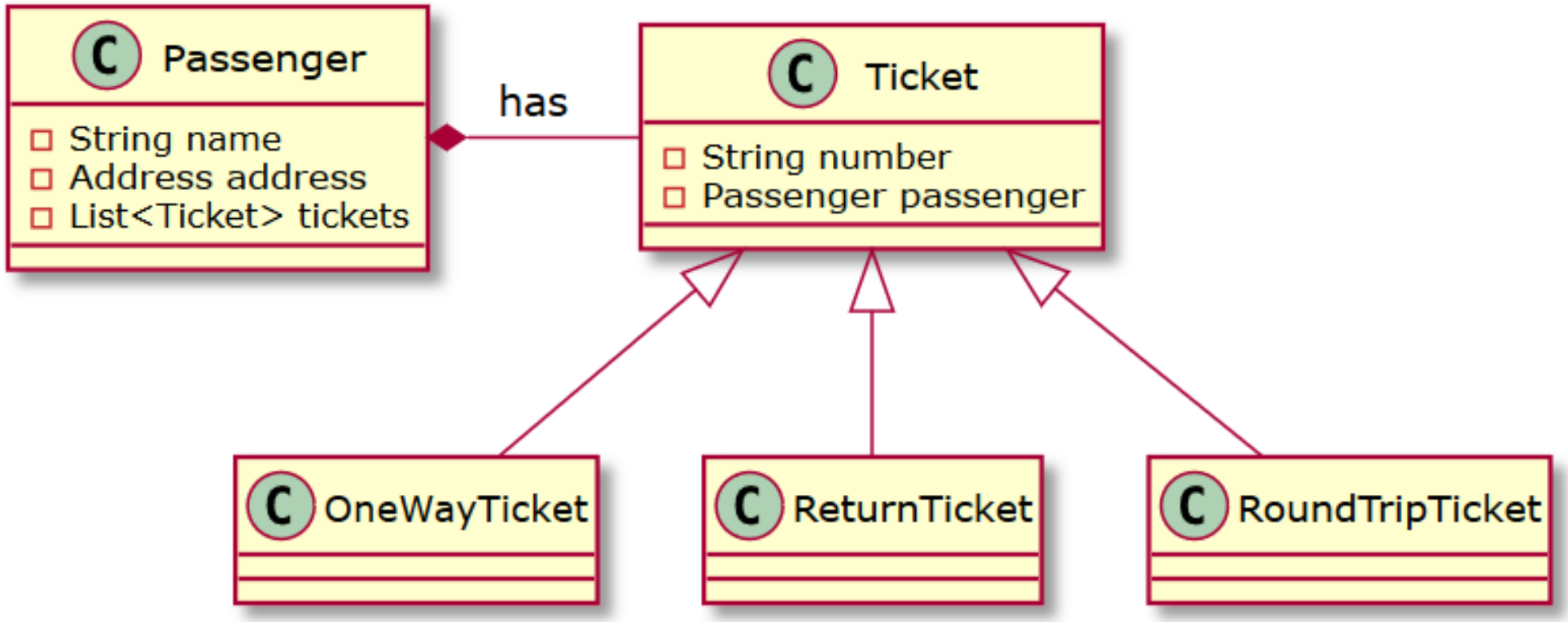


The Extended PASSENGERS Table

```
create table PASSENGERS (
        NAME varchar(255),
        ADDRESS_STREET varchar(30),
        ADDRESS_NUMBER varchar(6),
        ADDRESS_ZIPCODE varchar(10),
        ADDRESS_CITY varchar(25),
        ADDRESS_COUNTRY varchar(25),
        primary key (NAME)
)
```

The Inheritance Problem

Using Inheritance

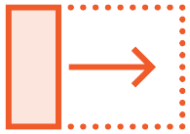


The Identity Problem

The PK in the PASSENGERS Table

```
create table PASSENGERS (
        NAME varchar(255),
        ADDRESS_STREET varchar(30),
        ADDRESS_NUMBER varchar(6),
        ADDRESS_ZIPCODE varchar(10),
        ADDRESS_CITY varchar(25),
        ADDRESS_COUNTRY varchar(25),
        primary key (NAME)
)
```

Defining Uniqueness



Objects identity (the == operator)

Logical equality (the equals method)

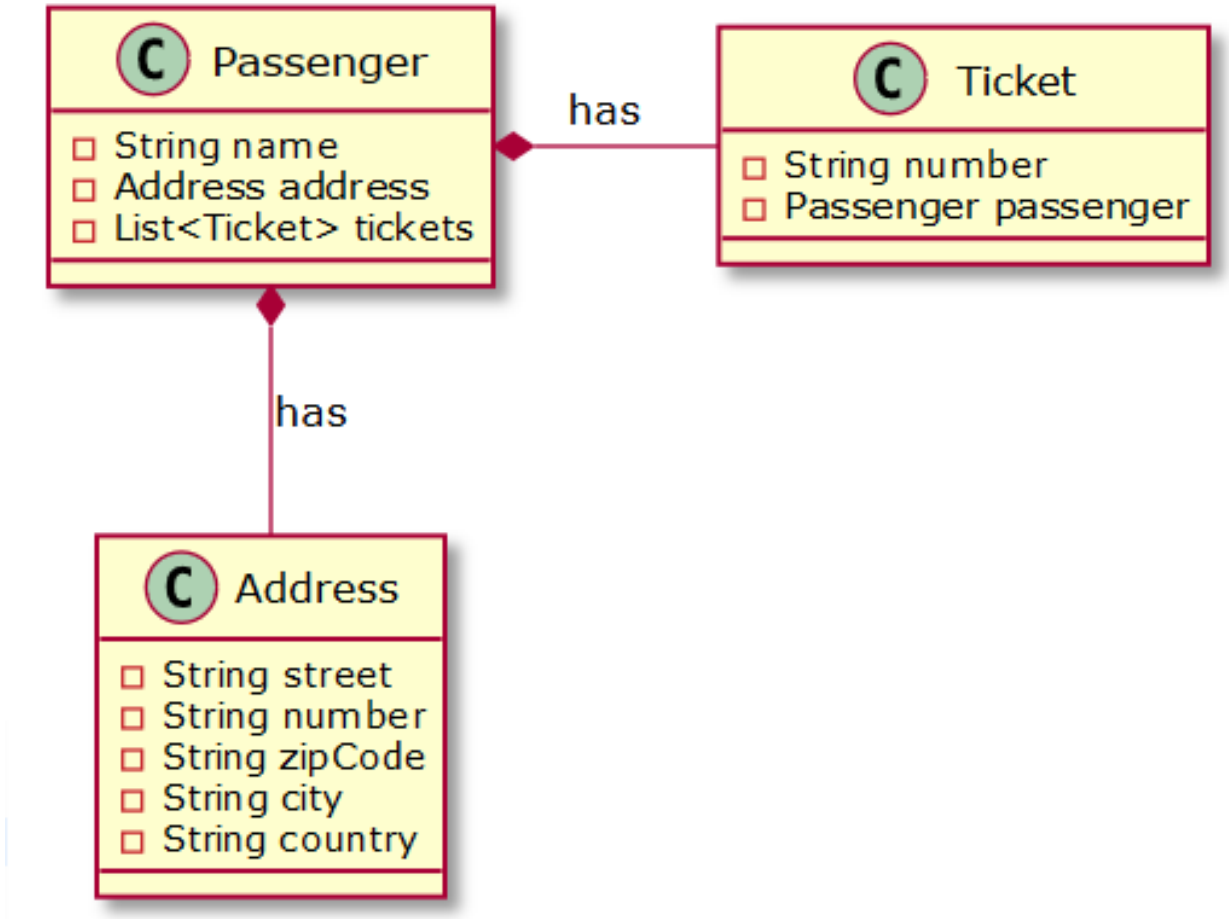
Primary keys

Tables with Surrogate Keys

```
create table PASSENGERS (
        ID integer not null,
        NAME varchar(255),
        primary key (ID)
)

create table TICKETS (
        ID integer not null,
        NUMBER varchar(255),
        PASSENGER_ID integer,
        primary key (ID)
)

alter table TICKETS
        add constraint FK_PASSENGERS
        foreign key (PASSENGER_ID)
        references PASSENGERS (ID)
```

The Associations Problem

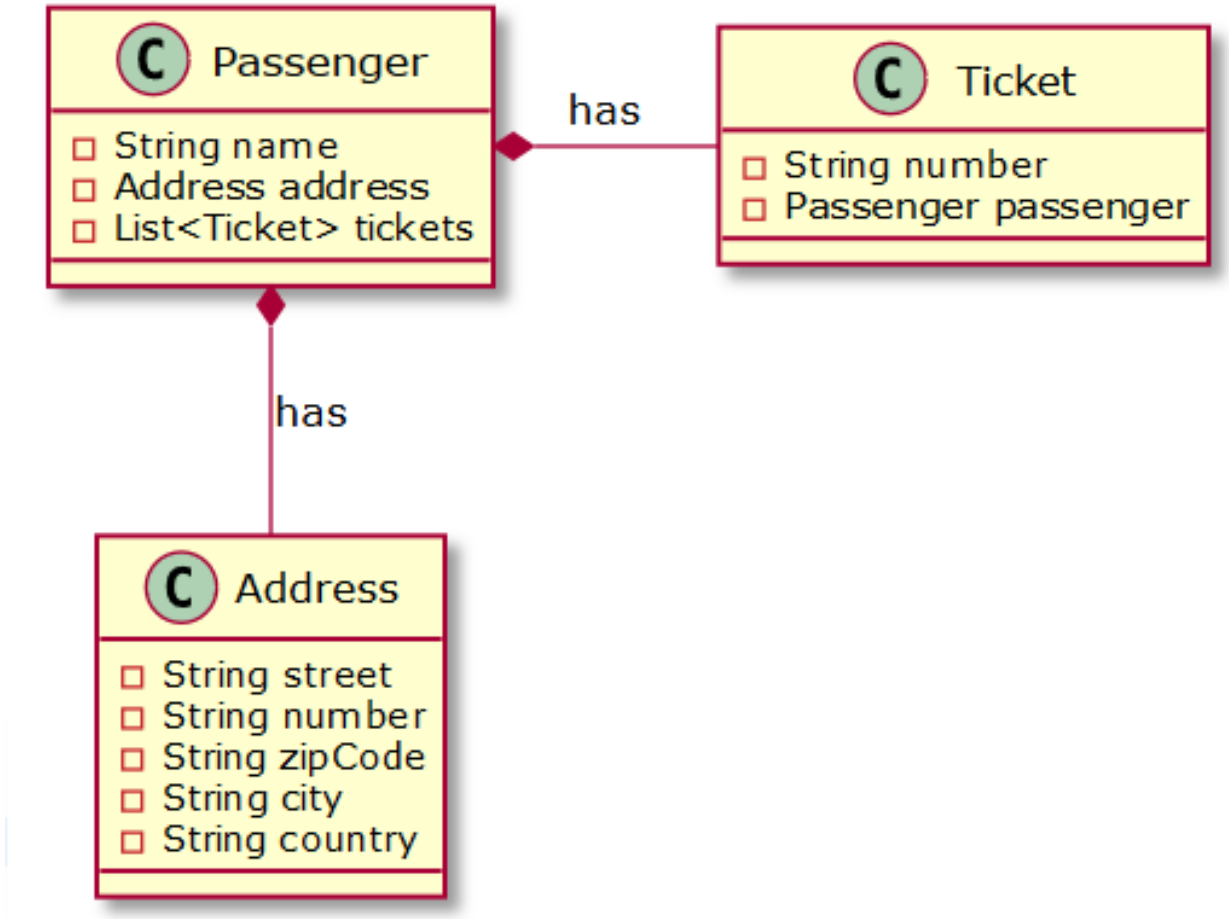
Associations in the Object-oriented Model



Associations in the Relational Model

```
create table PASSENGERS (
        ID integer not null,
        NAME varchar(255),
        ADDRESS_STREET varchar(30),
        ADDRESS_NUMBER varchar(6),
        ADDRESS_ZIPCODE varchar(10),
        ADDRESS_CITY varchar(25),
        ADDRESS_COUNTRY varchar(25),
        primary key (ID)
)

create table TICKETS (
        ID integer not null,
        NUMBER varchar(255),
        PASSENGER_ID integer,
        primary key (ID)
)
```

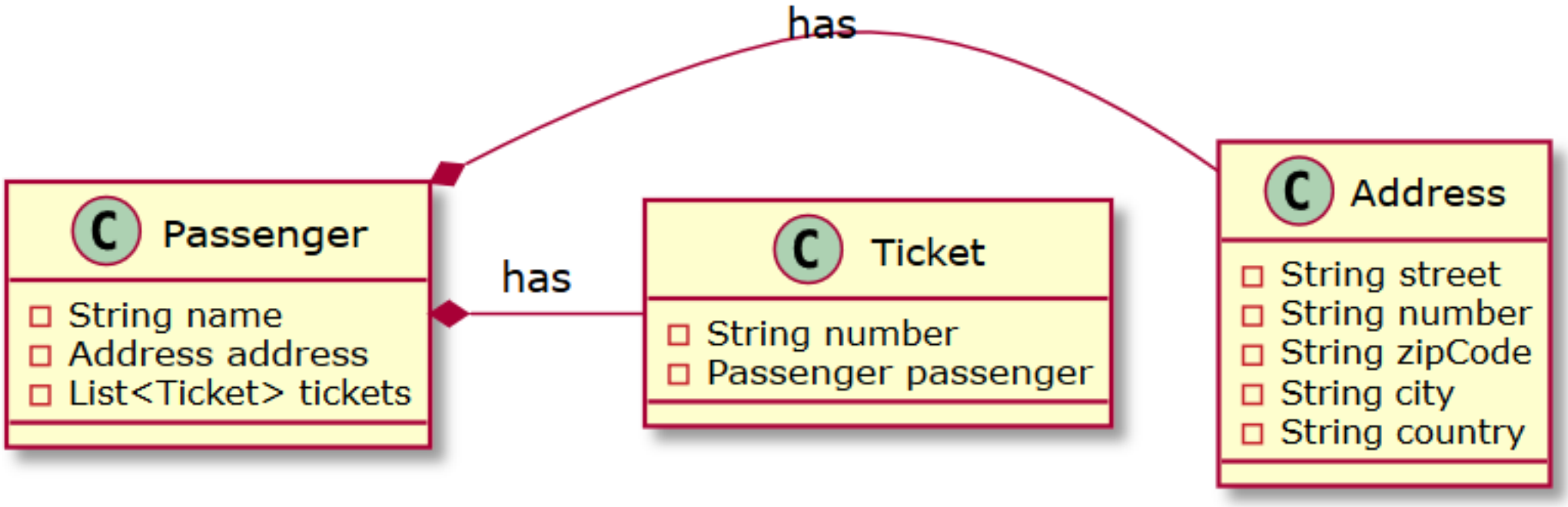
Associations in the Object-oriented Model



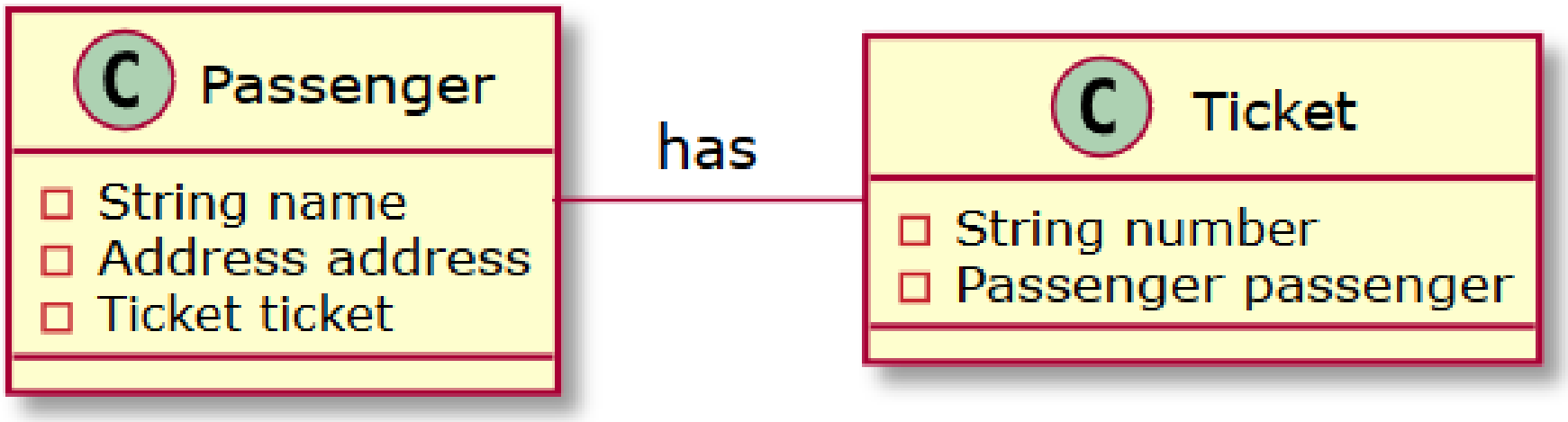
Associations in the Relational Model

```
create table PASSENGERS (
        ID integer not null,
        NAME varchar(255),
        ADDRESS_STREET varchar(30),
        ADDRESS_NUMBER varchar(6),
        ADDRESS_ZIPCODE varchar(10),
        ADDRESS_CITY varchar(25),
        ADDRESS_COUNTRY varchar(25),
        primary key (ID)
)

create table TICKETS (
        ID integer not null,
        NUMBER varchar(255),
        PASSENGER_ID integer,
        primary key (ID)
)
```

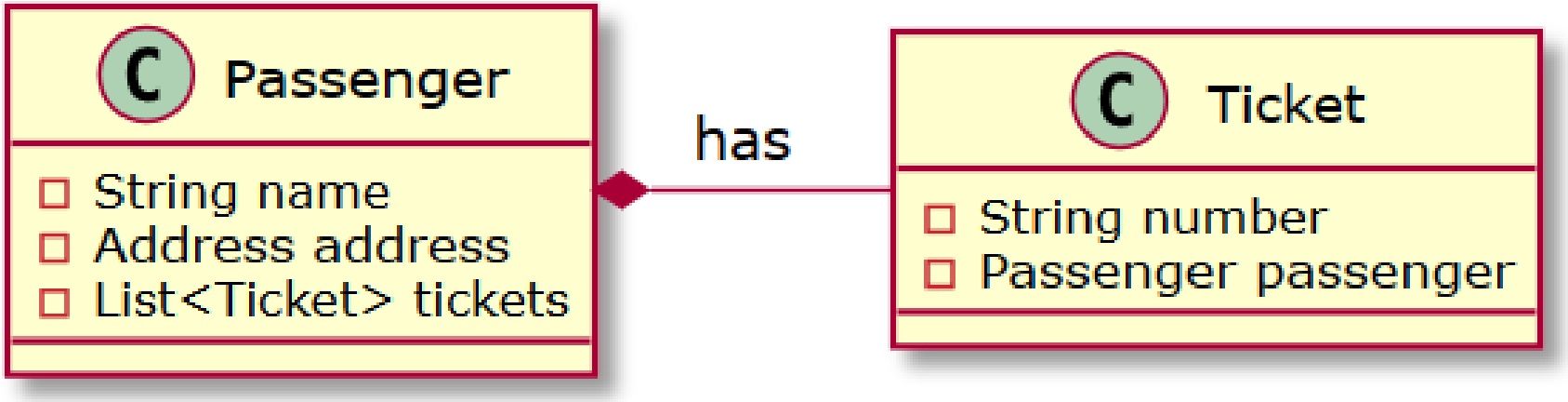
Associations in the Object-oriented Model



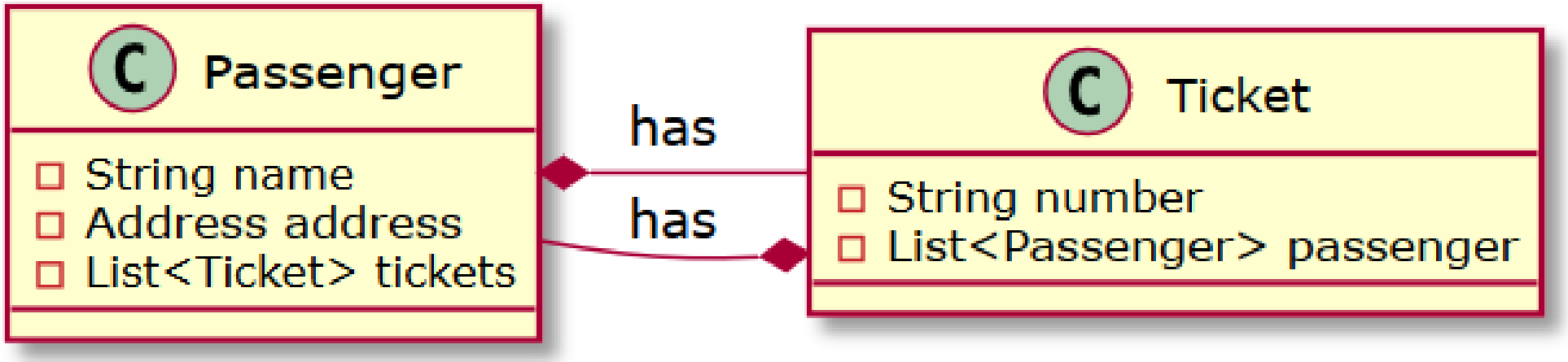
One-to-one Association



One-to-many Association



Many-to-many Association



Associations in the Relational Model

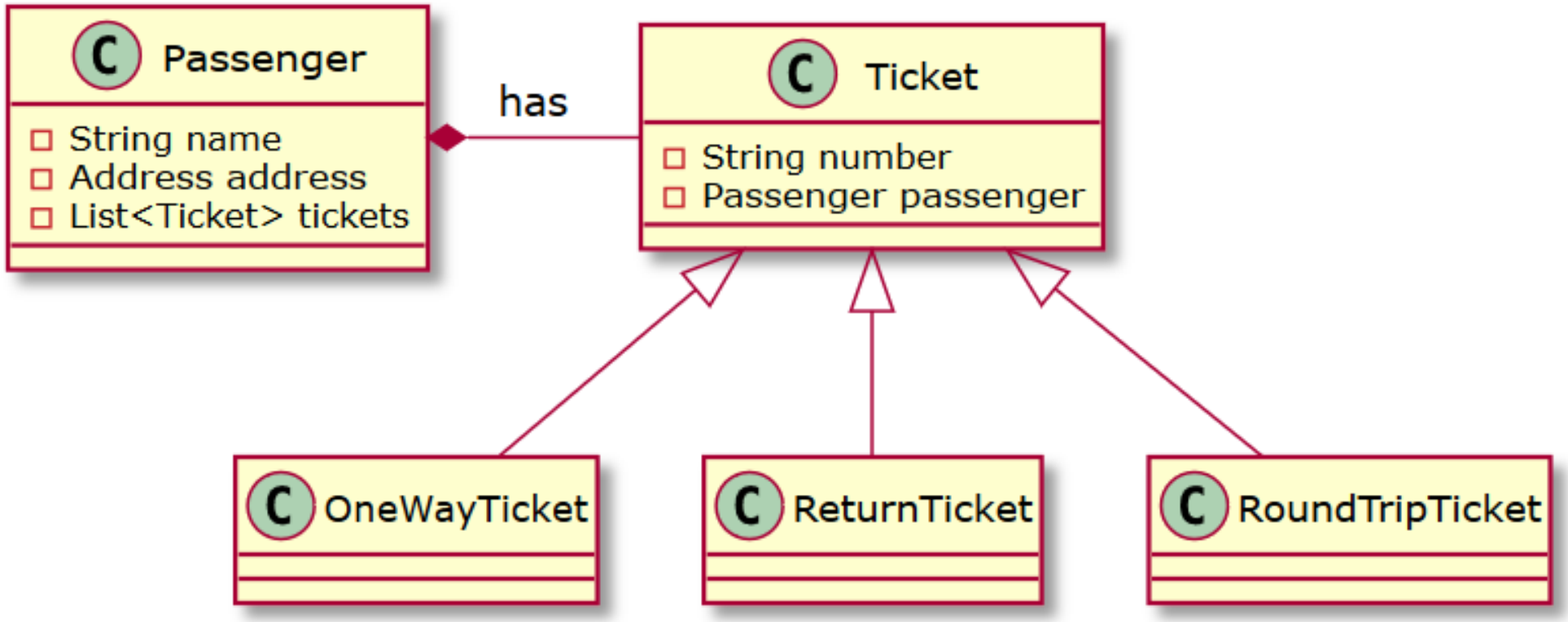
```
create table PASSENGERS (
        ID integer not null,
        NAME varchar(255),
        ADDRESS_STREET varchar(30),
        ADDRESS_NUMBER varchar(6),
        ADDRESS_ZIPCODE varchar(10),
        ADDRESS_CITY varchar(25),
        ADDRESS_COUNTRY varchar(25),
        primary key (ID)
)

create table TICKETS (
        ID integer not null,
        NUMBER varchar(255),
        PASSENGER_ID integer,
        primary key (ID)
)
```

Many-to-many Associations in Relational Model

```
create table PASSENGERS_TICKETS (
       PASSENGER_ID integer not null,
       TICKET_ID integer not null,
       primary key (PASSENGER_ID, TICKET_ID)
)

alter table PASSENGERS_TICKETS
       add constraint FK_PASSENGERS
       foreign key (PASSENGER_ID)
       references PASSENGERS (ID)

alter table PASSENGERS_TICKETS
       add constraint FK_TICKETS
       foreign key (TICKET_ID)
       references TICKETS (ID)
```

The Data Navigation Problem

Data Navigation in the Object-oriented Model



Data Navigation in the Relational Model

```
create table PASSENGERS (
        ID integer not null,
        NAME varchar(255),
        ADDRESS_STREET varchar(30),
        ADDRESS_NUMBER varchar(6),
        ADDRESS_ZIPCODE varchar(10),
        ADDRESS_CITY varchar(25),
        ADDRESS_COUNTRY varchar(25),
        primary key (ID)
)

create table TICKETS (
        ID integer not null,
        NUMBER varchar(255),
        PASSENGER_ID integer,
        primary key (ID)
)
```

Data Navigation Approaches

```
for(Ticket ticket: passenger.getTickets())
```

```
SELECT * FROM PASSENGERS WHERE ID = 727423
```

```
SELECT * FROM PASSENGERS, TICKETS
WHERE PASSENGERS.ID = 727423 AND
PASSENGERS.ID = TICKETS.PASSENGER_ID
```

Demo

Create a Hibernate Java project

Create the entity classes

Persist objects to the database

Summary

Object-relational Mapping (ORM)

Java Persistence API (JPA)

Hibernate:

- Advantages
- Drawbacks

Problems of Object-relational impedance mismatch

Simple Hibernate application

Working with Entities

Cătălin Tudose

PHD IN COMPUTER SCIENCE, JAVA AND WEB TECHNOLOGIES EXPERT

www.catalintudose.com
https://www.linkedin.com/in/catalin-tudose-847667a1

Overview

What is an entity?

How to map objects

Entity access types

Define entity primary keys and entity identity

What Is an Entity?

Annotated with @Entity or defined through XML configuration

Top-level class

Public or a protected no-arguments constructor

Final classes not recommended

What Does an Entity Support?

Inheritance

Polymorphic associations

Polymorphic queries

The Persistent State

Primitive types

Entity types

Embeddable types

Serializable types

Enums

Collections

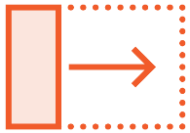
Mapping the Objects

@Table

@SecondaryTable

@SecondaryTables

The @Table Annotation

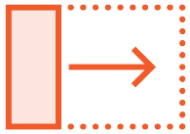


Primary table

Default values may apply

Parameters

The @SecondaryTable Annotation

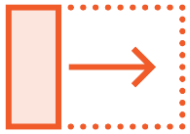


Secondary table

Parameters

pkJoinColumns and foreignKey

The @SecondaryTables Annotation

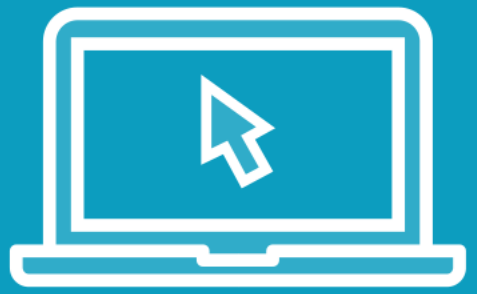


Multiple secondary tables

Multiple @SecondaryTable annotations as parameters

The same or differently named primary keys

Demo



A secondary table with one field

Passenger with an address

- The address has one field

Demo

A secondary table with multiple fields

Passenger with an address

- The address has multiple fields

Demo

Multiple secondary tables

Passenger with address and phone

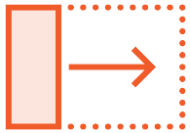
Entity Access Types

Accessing the Persistent State

Field access

Property access

Entity Access Type



Method to access the persistent state of the entity

Single access type by default

Determined by placing the mapping annotations

Entity Access Type

```
@Entity
@Table(name = "PASSENGERS")
public class Passenger {
    @Id
    @Column(name = "PASSENGER_ID")
    private int id;

    @Column(name = "PASSENGER_NAME", table = "PASSENGERS")
    private String name;
```

Entity Access Type

```
@Entity
@Table(name = "PASSENGERS")
public class Passenger {
    private int id;

    @Id
    @Column(name = "PASSENGER_ID")
    public int getId() {
        return id;
    }
```

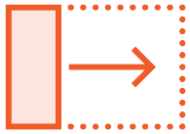
Entity Access Type

```
@Entity
@Table(name = "PASSENGERS")
public class Passenger {
    @Id
    private int id;

    private String name;
```

Entity Access Type

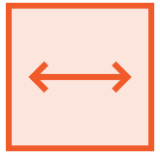
```
@Entity
@Table(name = "PASSENGERS")
public class Passenger {
    private int id;

    @Id
    public int getId() {
        return id;
    }
```

@Access(AccessType.FIELD)



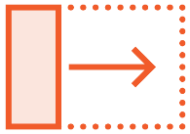
Default access type for the class is field-based

Persistent state accessed via the instance variables



@Access(AccessType.PROPERTY) for needed properties

@Access(AccessType.PROPERTY)



Default access type for the class is property-based

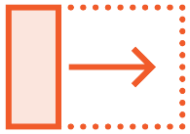
Persistent state accessed via the properties

@Access(AccessType.FIELD) for needed instance variables

Mixing Access Types

```
@Entity
@Table(name = "PASSENGERS")
@Access(AccessType.FIELD)
public class Passenger {
    @Id
    private int id;
    private String name;

    public int getId() {
        return id;
    }

    @Access(AccessType.PROPERTY)
    public String getName() {
        return name;
    }
```

Field-based vs. Property-based Access



Omit the getter methods for the fields that should not be exposed

Easier readability using field-based access

Additional logic through accessors

Entity Primary Keys and Entity Identity

Defining Primary Keys

Simple primary key

Composite primary key

Rules for Composite Primary Keys

Public class, public no-arguments constructor

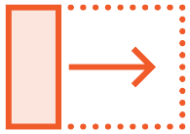
Serializable

equals() and hashCode()

Represented as embeddable class or as an id class

Fields or properties correspondence

The @GeneratedValue Annotation



Generation strategy

Used in conjunction with @Id

May be applied to a persistent field or property

Demo

Primary key

@GeneratedValue

Demo

Primary key

Embeddable primary key/embedded id

Demo

Primary key

Embeddable primary key/id class

Summary

Entities

Mapping objects

Entity access types

- Field-based
- Property-based

Entity primary keys and entity identity

- @GeneratedValue
- Embeddable primary key/embedded id
- Embeddable primary key/id class

Entity Relationships

Cătălin Tudose

PHD IN COMPUTER SCIENCE, JAVA AND WEB TECHNOLOGIES EXPERT

www.catalintudose.com
https://www.linkedin.com/in/catalin-tudose-847667a1

Overview

The types of relationships

The directions of the relationships

Annotations used for relationships

Embedded classes

Collections of embedded classes

Types of Relationships

One-to-many

Many-to-one

Many-to-many

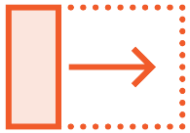
One-to-one

Direction of the Relationship

Unidirectional

Bidirectional

Owning Side

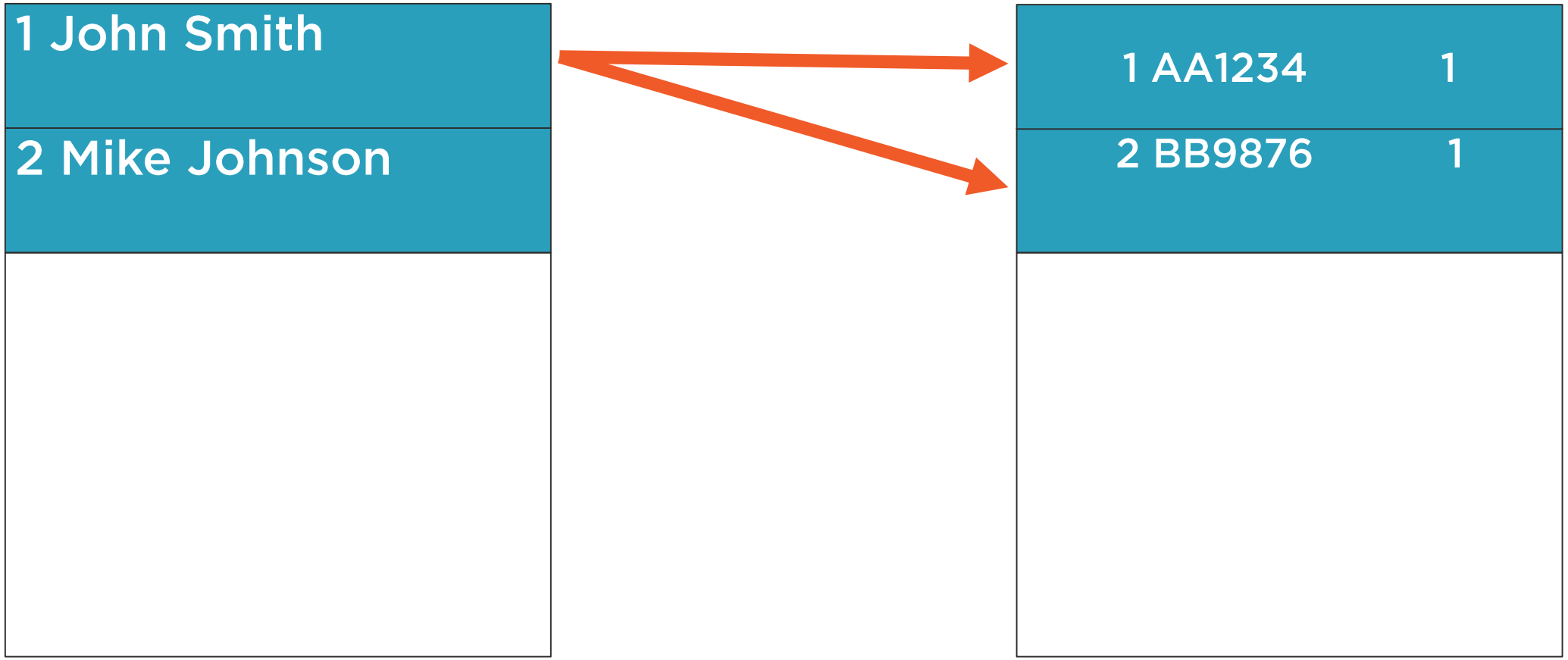


Any relationship has an owning side

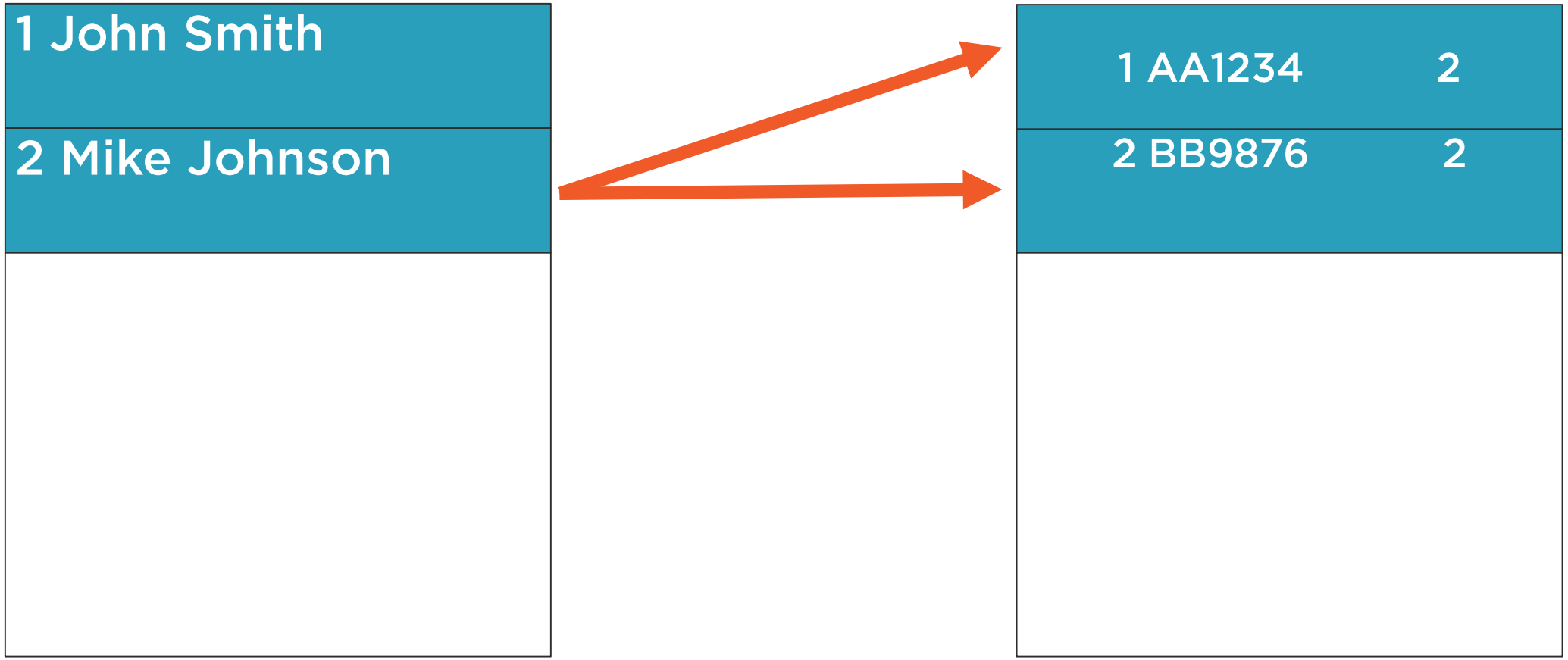
Owning side drives the updates to relationship in a database

The many part is always the owning side

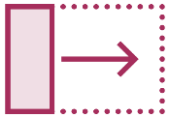
Changing the Relationship



Changing the Relationship



Annotations to Relationships



@OneToMany

@ManyToOne

@OneToOne

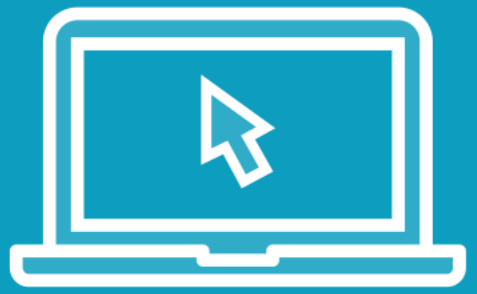
@ManyToMany

Demo

Define one-to-many relationships

Define many-to-one relationships

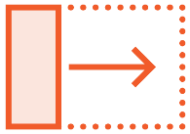
Demo



Define many-to-many relationships

Annotations for Relationship Definition

Annotations for Relationship Definition

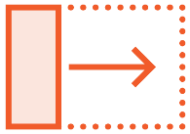


@JoinTable

@JoinColumn

@JoinColumns

@JoinTable

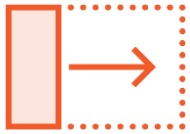


Cross-reference table for the mapping of relationship

On the owning side of relationship

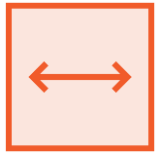
Parameters

@JoinColumn



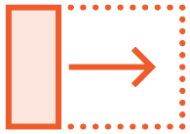
Specifies a column for joining an entity association

Name parameter



referencedColumnName parameter

@JoinColumns



Mapping for composite foreign keys

Groups @JoinColumn annotations

Value parameter

Demo

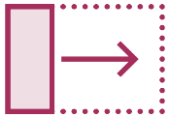
Join tables on one column

Demo

Join tables on multiple columns

Embeddable Classes

Embeddable Classes



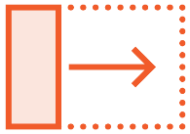
Fine-grained classes representing entity state

Do not have persistent identity of their own

Cannot be shared across persistent entities

Only as part of the state of the entity to which they belong

Embeddable Classes Annotations

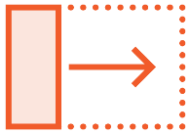


@Embeddable

@AttributeOverride

@AttributeOverrides

Collections of Embeddable Classes

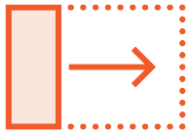


Collections of basic types or embeddable classes

@ElementCollection

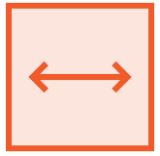
@CollectionTable

Maps of Embeddable Classes



Key and value: either a basic type or an embeddable class

@MapKeyColumn



@Column

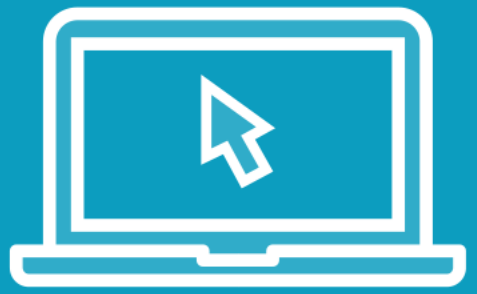
Demo

Embedding classes in entities

Demo

Embedding collections of classes in entities

Demo



Embedding maps of classes in entities

Summary

- Types and directions of relationships
- Annotations to define relationships
- Embedded classes
- Collections of embedded classes
- Maps of embedded classes

Entity Inheritance

Cătălin Tudose
PHD IN COMPUTER SCIENCE, JAVA AND WEB TECHNOLOGIES EXPERT

www.catalintudose.com
https://www.linkedin.com/in/catalin-tudose-847667a1

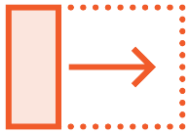
Overview

Inherit from entities and non-entities

Working with mapping strategies

Working with converters

Abstract Entity Classes

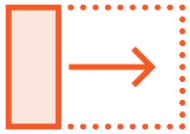


Annotated with the @Entity annotation

Mapped as an entity

Target of queries

Entities Inheriting from Non-entities

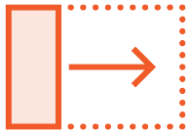


Define state and mapping information common to multiple entities

The mapped superclass cannot be queried

Only unidirectional relationships

Annotations to Inherit from Non-entities

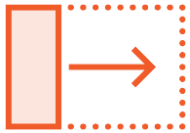


@MappedSuperclass

@AttributeOverride

@AssociationOverride

Extending Non-entities with Non-persistent State



Non-persistent state

Inherit behavior

Annotations are ignored

Demo

Extend one entity

Demo

Extend one non-entity

Mapping Strategies

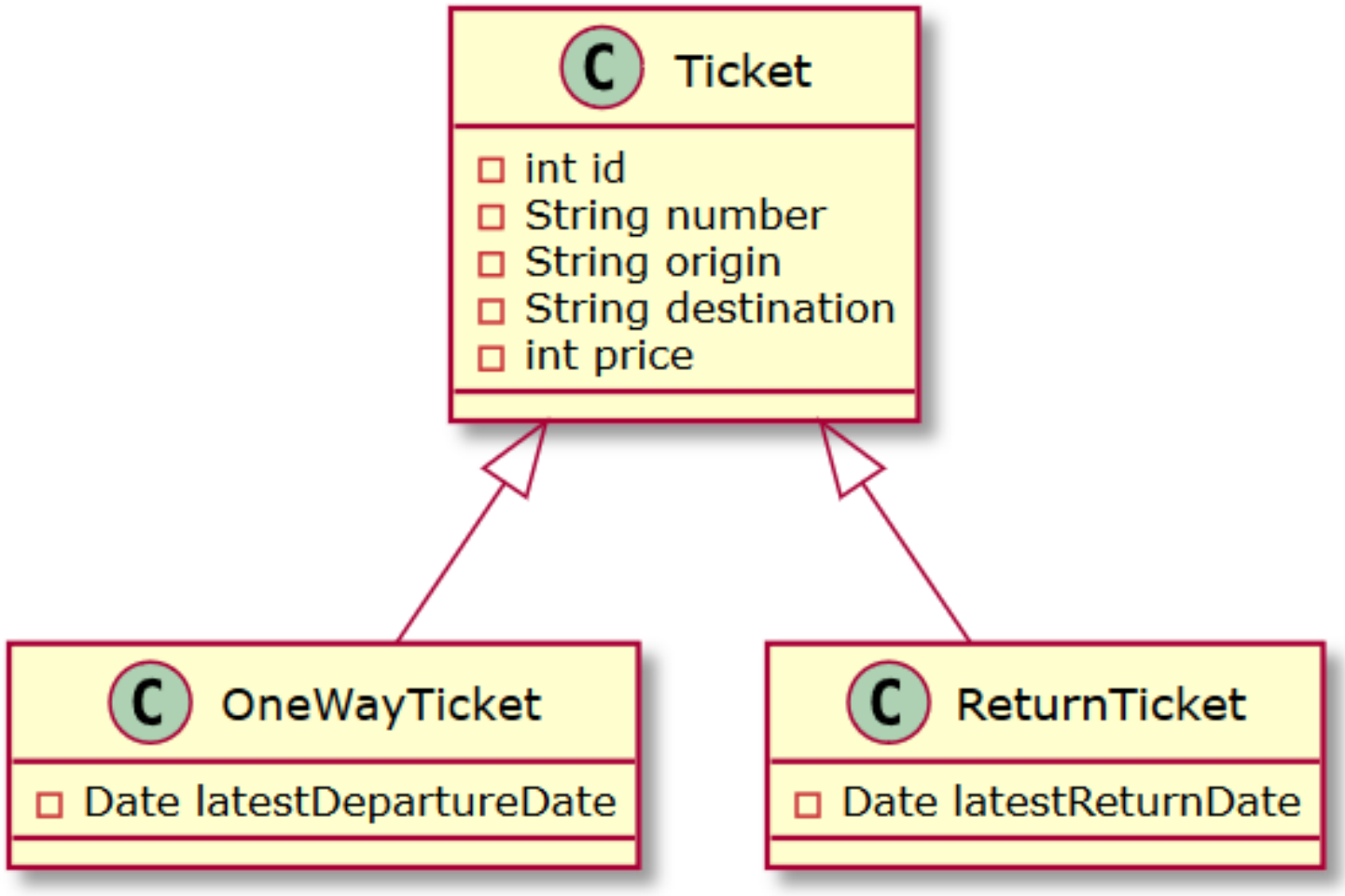
Mapping Strategies

Single table per class hierarchy

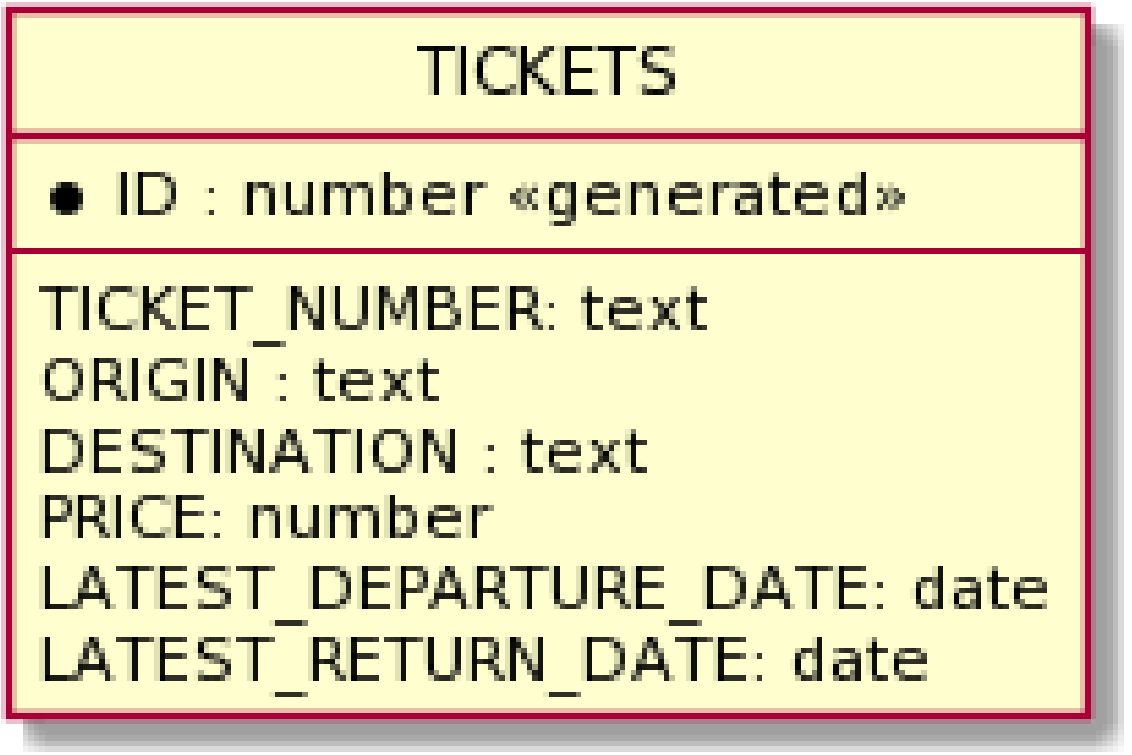
Joined subclass strategy

Table per concrete entity class

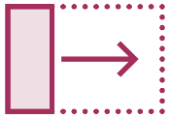
The Classes Hierarchy



Single Table per Class Hierarchy



Single Table per Class Hierarchy



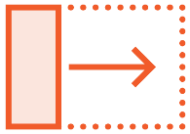
All classes in the hierarchy mapped to a single table

Discriminator column

Good support for polymorphic relationships

Requires nullable columns

Annotations for Mapping Strategies

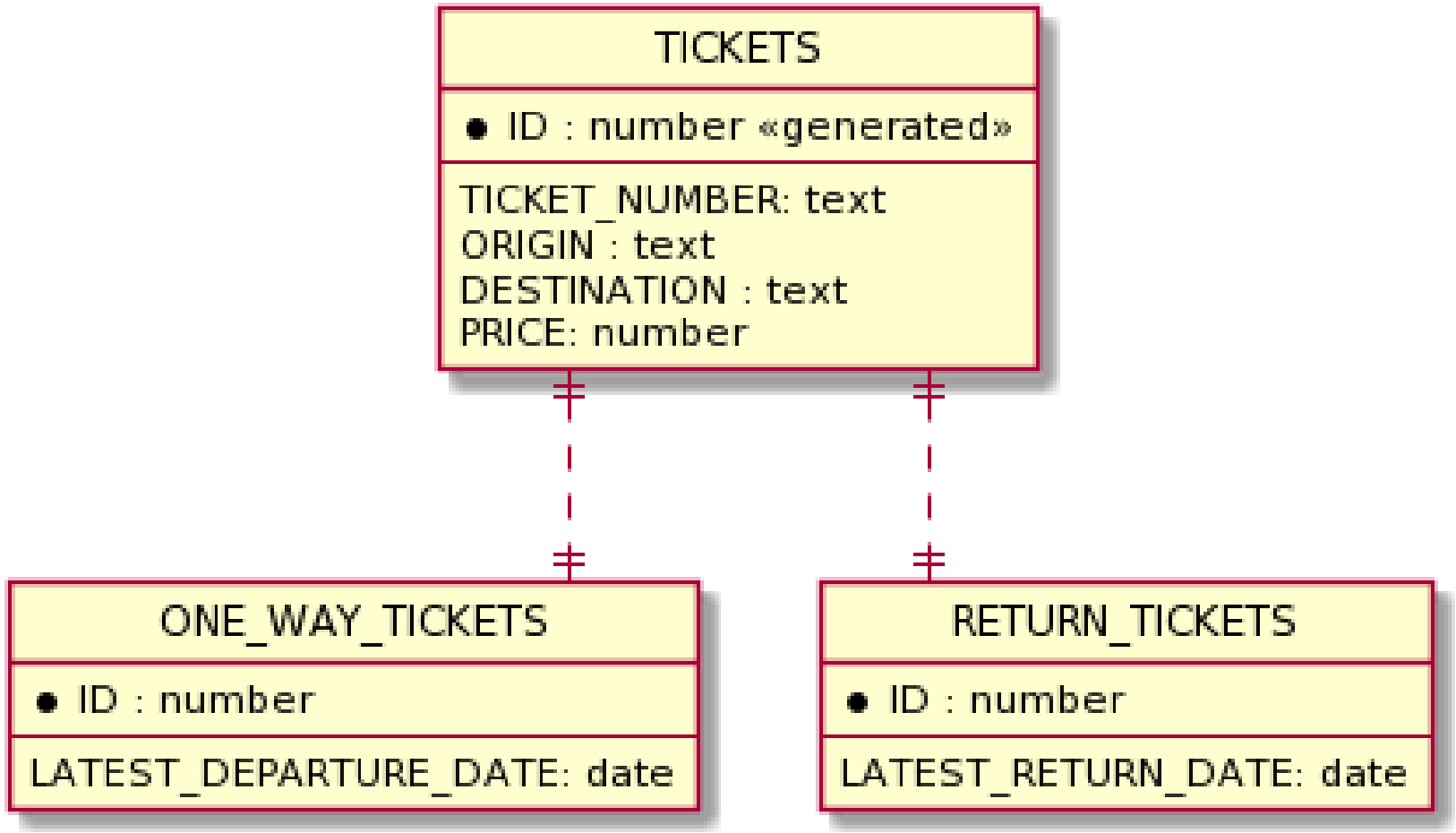


@Inheritance

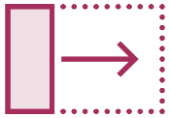
@DiscriminatorColumn

@DiscriminatorValue

Joined Subclass Strategy



Joined Subclass Strategy



Classes as separate tables

Join through the primary key column

Good support for polymorphic relationships

More join operations between entities

Table per Concrete Class Strategy

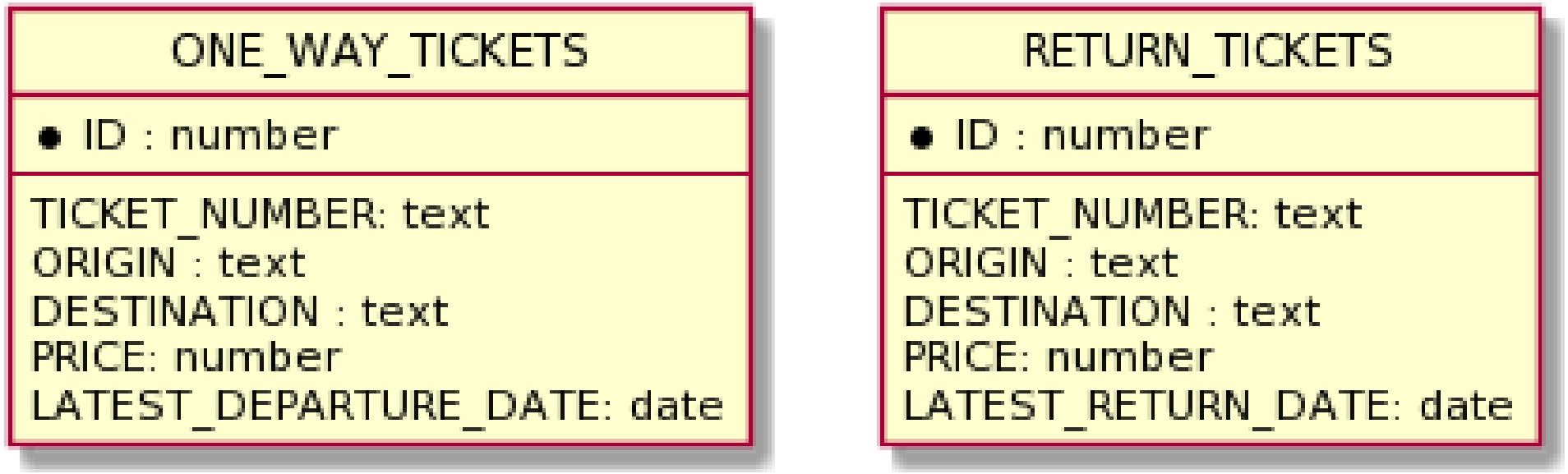
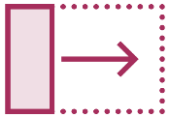


Table per Concrete Class Strategy



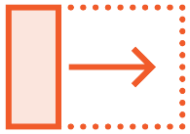
Each subclass is mapped to a separate table

All properties mapped to columns of the table for the class

Poor support for polymorphic relationships

May duplicate the IDs between tables

Storing Values Represented Differently

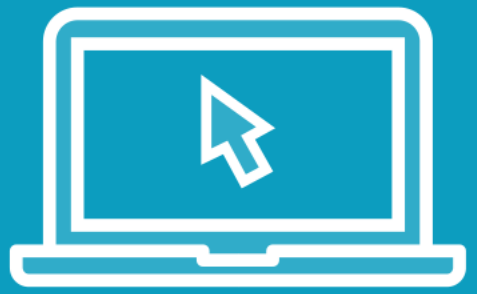


Java boolean ⇔ 0/1, True/False, Yes/No

javax.persistence.AttributeConverter

Convert entity attribute state to database column and vice-versa

Demo



Single table per class hierarchy

Demo

Joined subclass strategy

Demo

Table per concrete class strategy

Demo

Conversion

Summary

Build class hierarchies

- Inherited from entities
- Inherited from non-entities

Applied the mapping strategies

- Single table per class hierarchy
- Joined subclass strategy
- Table per concrete class strategy
- Particularities, pluses, minuses

Worked with converters

The EntityManager API

Cătălin Tudose

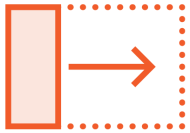
PHD IN COMPUTER SCIENCE, JAVA AND WEB TECHNOLOGIES EXPERT

www.catalintudose.com
https://www.linkedin.com/in/catalin-tudose-847667a1

Overview

- Interfaces and classes for persistence
- Methods of the EntityManager interface
- The states of the entity instance
- Commit and flush
- Application/container managing an EntityManager

What Is an EntityManager?

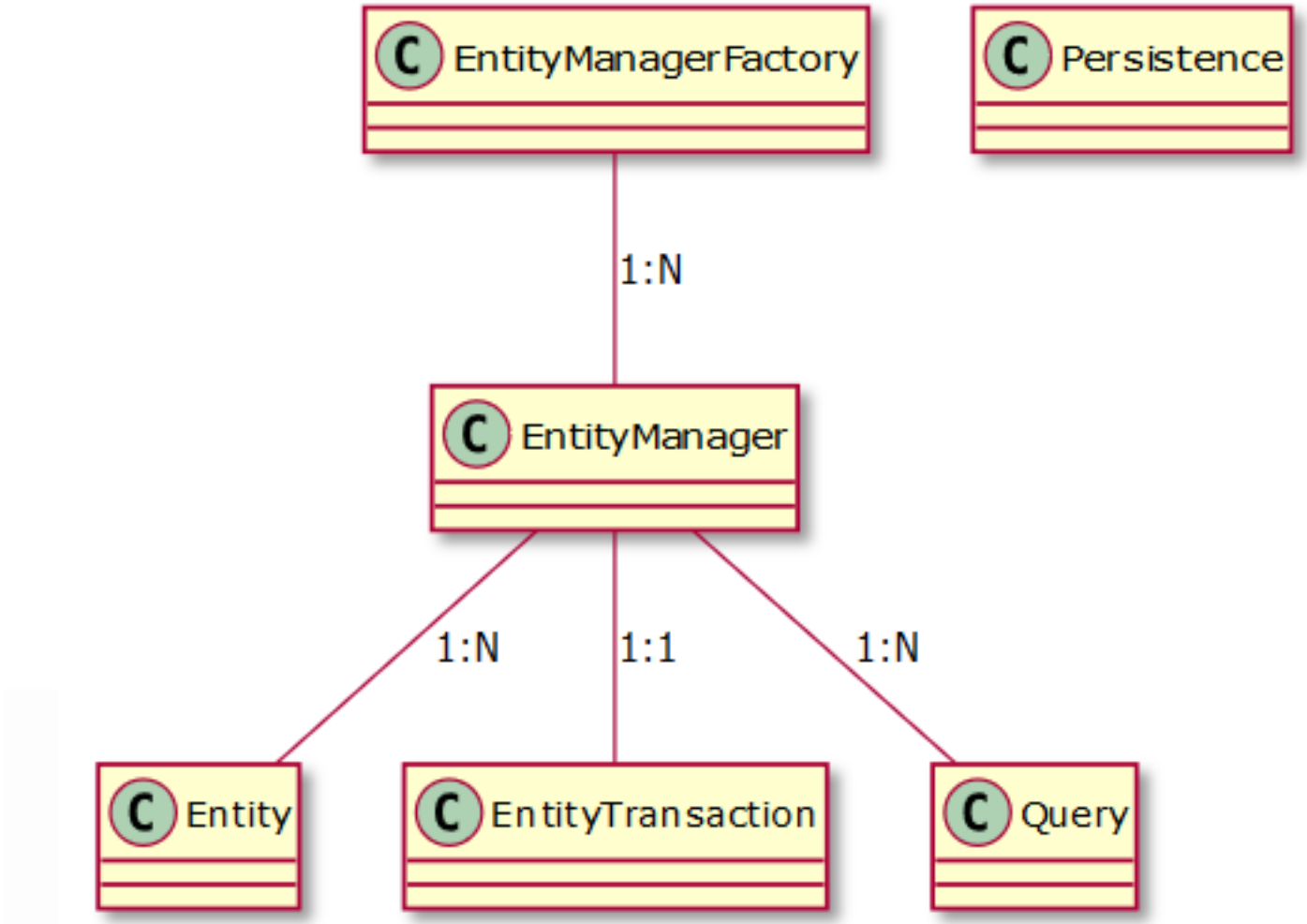


An interface with methods to interact with the persistence context

Persistence context

Persistence unit

The Persistence API Classes



Entity States

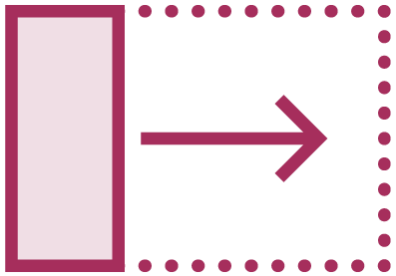
New

Managed

Removed

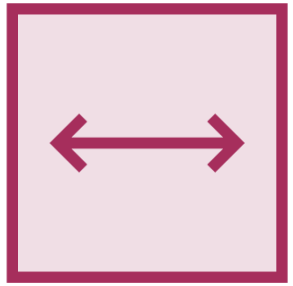
Detached

The EntityManager Methods



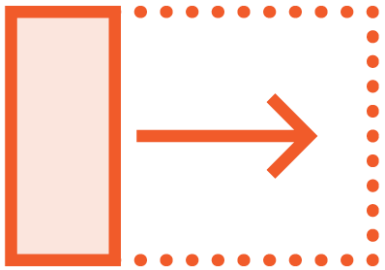
`persist()`

`remove()`



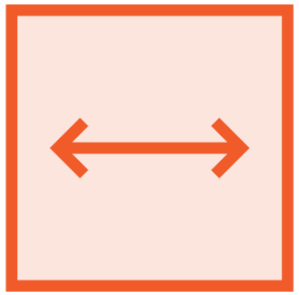
`merge()`

`find()`



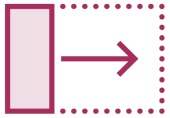
`lock()`

`detach()`



`refresh()`

Using the `persist` Method



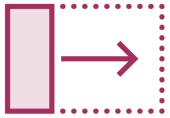
New => Managed

Managed => Ignored, cascaded to referenced entities

Detached => Exception

Removed => Managed

Using the `merge` Method



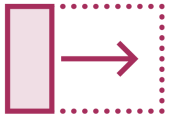
New => New instance with the same state created

Managed => Ignored, cascaded to referenced entities

Detached => Existing entity loaded; state merged

Removed => IllegalArgumentException

Using the `remove` Method



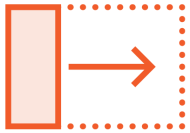
New => Ignored

Managed => Removed, cascaded to referenced entities

Detached => IllegalArgumentException

Removed => Ignored

Using the `refresh` Method

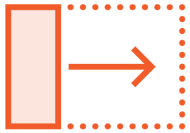


Managed => Entity reloaded from the database

Referenced entities cascaded

New/detached/removed => IllegalArgumentException

Synchronize the Database with `commit` and `flush`



Commit the transaction or flush the operation

`commit` **– ends the unit of work, no rollback**

`flush` **– normal synchronization, no commit**

Obtaining an Application-managed EntityManager

EntityManagerFactory
=> EntityManager

Application-managed
EntityManager

Programmatically
control the lifecycle

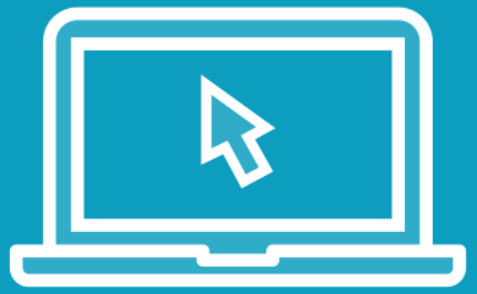
Obtaining a Container-managed EntityManager

Dependency injection

Spring Framework

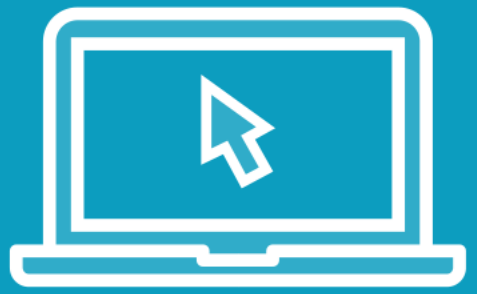
@PersistenceContext

Demo



Application-managed EntityManager

Demo



Container-managed EntityManager

Summary

Interfaces and classes for persistence

Methods of the EntityManager interface

- Persist, remove, merge, find
- Lock, detach, refresh

The states of the entity instance

- New, managed, detached, removed

Commit and flush

Application managing an EntityManager

Container managing an EntityManager, using Spring

Course Summary

- ORM and its problems
- Working with entities
- Entity relationships
- Entity inheritance
- The EntityManager API

Related Courses

Spring Framework: Spring Fundamentals - Bryan Hansen

Spring with JPA and Hibernate - Bryan Hansen

TDD with JUnit 5 - Cătălin Tudose