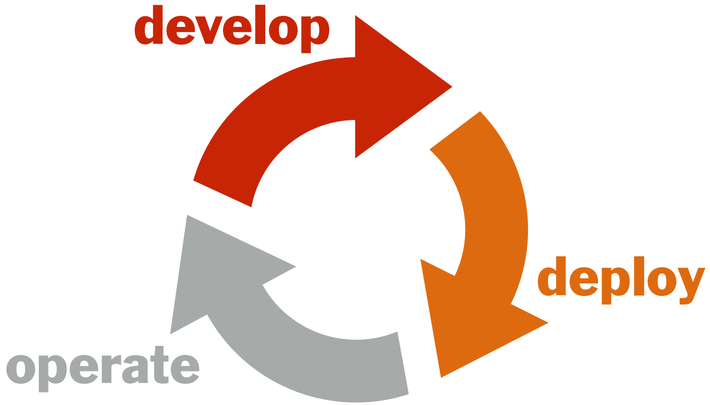
object persistence

& transactions

learning objectives



- learn about persistent objects
- learn about object serialization
- learn about object-relational mapping
- learn about transaction management

what are persistent objects?

there exists two types of memory in a computer:

- volatile memory ➡ its content disappears when the computer shuts down or restarts
- non-volatile memory ➡ its content persists when the computer shuts down or restarts

by default, objects reside in volatile memory

they are lost when the computer shuts down or restarts

a persistent object exists primarily in non-volatile memory but also in volatile memory when it is used*

*invoked

what are persistent objects?

there exists various approaches to persist objects:

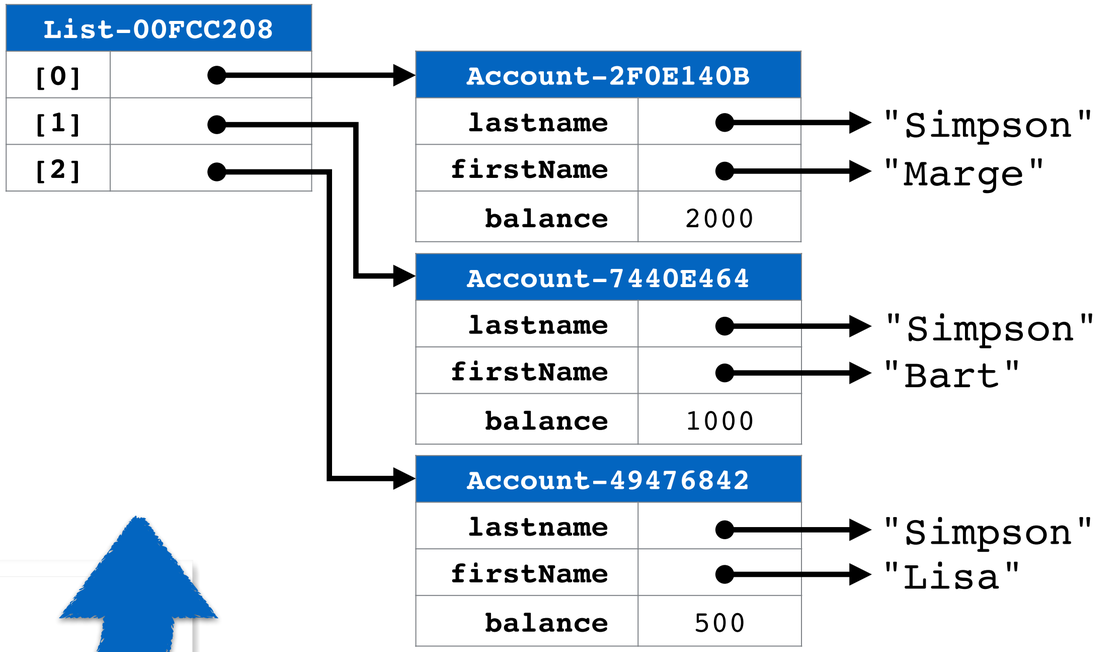
- serializing objects to a file ➡ objects are encoded as a stream stored into a file
  - in some xml or json or binary format
- mapping objects to a database ➡ objects are mapped to a relational database
- replicating object across the network ➡ multiple replicas of each object are kept in the volatile memory of different computers, so that there always exists one replica of that object in the volatile memory of at least one computer

this approach comes at the cost of synchronizing the replicas of each object to keep them consistent

serializing objects to a file

```
public class Account {
    private String lastName;
    private String firstName;
    private double balance = 0.0;

    public Account(String lastName, String firstName, double balance) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    …
    @Override
    public String toString() {
        return getClass().getSimpleName() + "-" +
        String.format("%08X", this.hashCode())  +
        "[" + lastName + ", " + firstName + ", $" + balance +"]";
    }
}
```




```
public class Main {
    public static void main(String args[]) throws FileNotFoundException,
                                                   IOException, ClassNotFoundException {

        List<Account> accounts = List.of(new Account("Simpson", "Marge", 2000),
                                         new Account("Simpson", "Bart", 1000),
                                         new Account("Simpson", "Lisa", 500));

        System.out.println("List-" + String.format("%08X", accounts.hashCode()) + accounts);
    }
}
```

1

```
List-00FCC208[Account-2F0E140B[Simpson, Marge, $2000.0], Account-7440E464[Simpson, Bart, $1000.0], Account-49476842[Simpson, Lisa, $500.0]]
```

serializing objects to a file

```java
import java.io.Serializable;

public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    private String lastName;
    private String firstName;
    private double balance = 0.0;

    public Account(String lastName, String firstName, double balance) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    …
    @Override
    public String toString() {
        return getClass().getSimpleName() + "-" +
        String.format("%08X", this.hashCode())  +
        "[" + lastName + ", " + firstName + ", $" + balance +"]";
    }
}
```

```
tmp — -bash — ttys001
wallace-palace:tmp garbi$ pwd
/tmp
wallace-palace:tmp garbi$ ls -l accounts.ser
-rw-r--r--  1 garbi  wheel  128 Oct 21 16:22 accounts.ser
wallace-palace:tmp garbi$ file accounts.ser
accounts.ser: Java serialization data, version 5
wallace-palace:tmp garbi$
```

```java
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Main {
    public static void main(String args[]) throws FileNotFoundException,
                                                   IOException, ClassNotFoundException {

        List<Account> accounts = List.of(new Account("Simpson", "Marge", 2000),
                                         new Account("Simpson", "Bart", 1000),
                                         new Account("Simpson", "Lisa", 500));

        System.out.println("List-" + String.format("%08X", accounts.hashCode()) + accounts);

        FileOutputStream fos = new FileOutputStream("/tmp/accounts.ser");
        try (ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(accounts);
        }

        FileInputStream fis = new FileInputStream("/tmp/accounts.ser");
        try (ObjectInputStream ois = new ObjectInputStream(fis)) {
            accounts = (List<Account>) ois.readObject();
        }

        System.out.println("List-" + String.format("%08X", accounts.hashCode()) + accounts);
    }
}
```

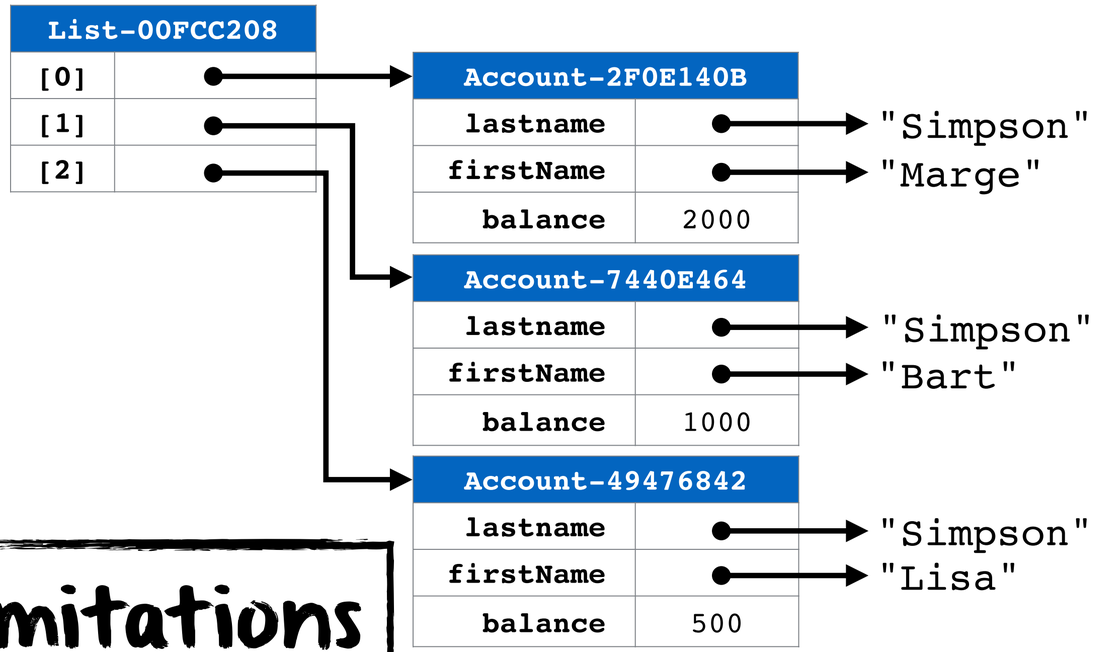
write to file

read from file

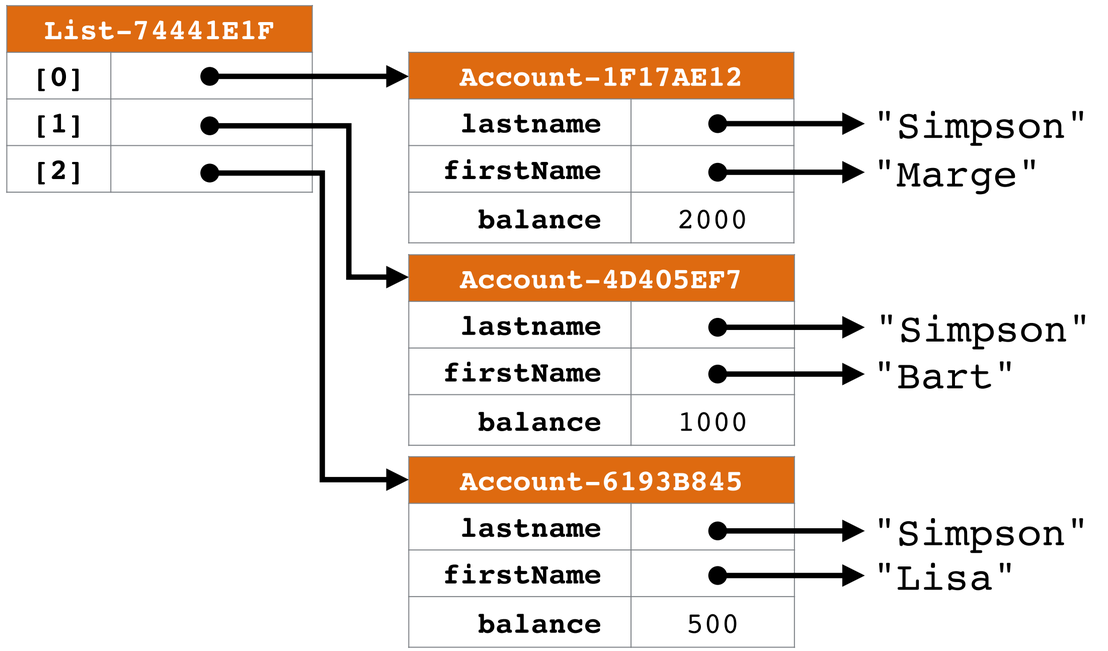
```
List-00FCC208[Account-2F0E140B[Simpson, Marge, $2000.0], Account-7440E464[Simpson, Bart, $1000.0], Account-49476842[Simpson, Lisa, $500.0]]
List-74441E1F[Account-1F17AE12[Simpson, Marge, $2000.0], Account-4D405EF7[Simpson, Bart, $1000.0], Account-6193B845[Simpson, Lisa, $500.0]]
```

serializing objects to a file

```
List-00FCC208[Account-2F0E140B[Simpson, Marge, $2000.0],
Account-7440E464[Simpson, Bart, $1000.0],
Account-49476842[Simpson, Lisa, $500.0]]
```




```
List-74441E1F[Account-1F17AE12[Simpson, Marge, $2000.0],
Account-4D405EF7[Simpson, Bart, $1000.0],
Account-6193B845[Simpson, Lisa, $500.0]]
```

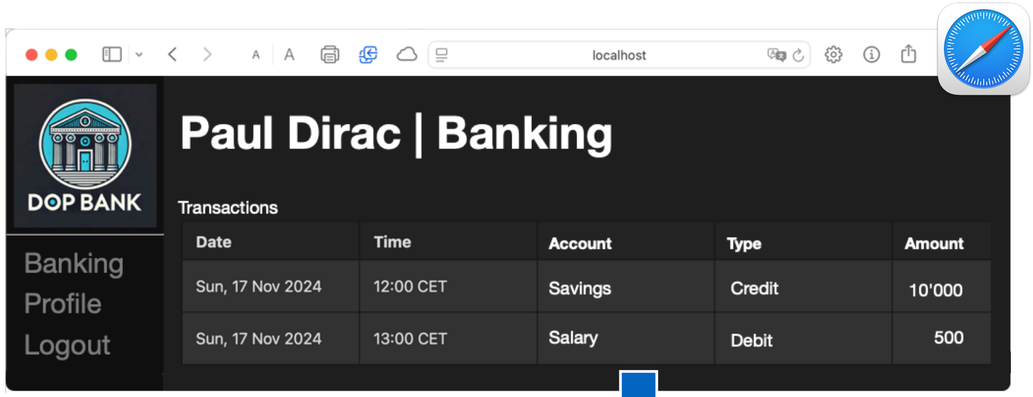


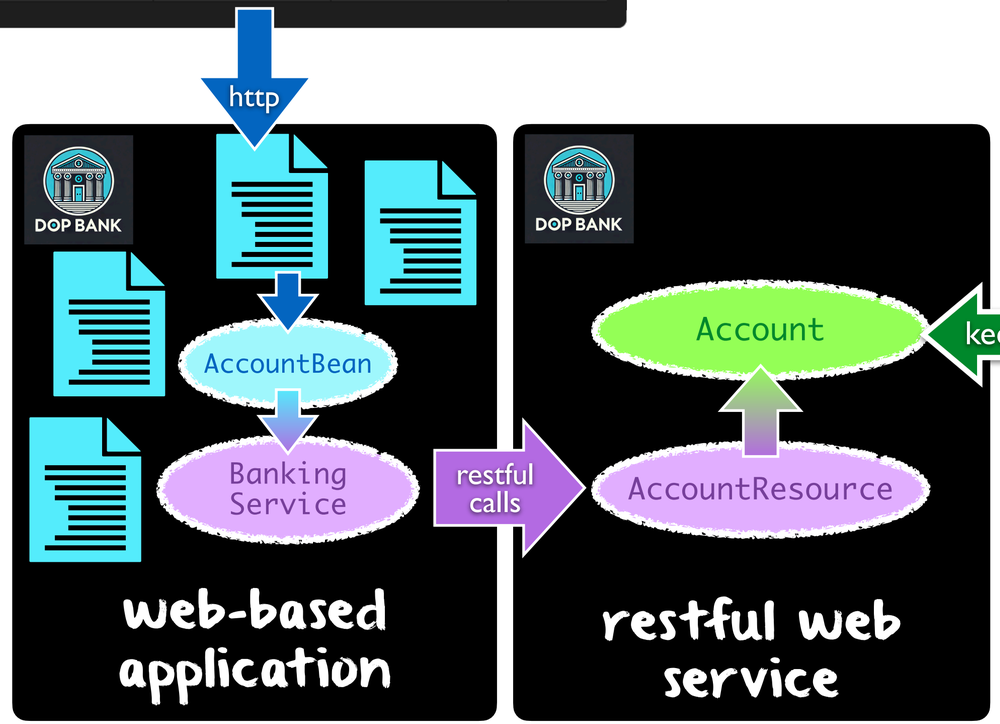
limitations

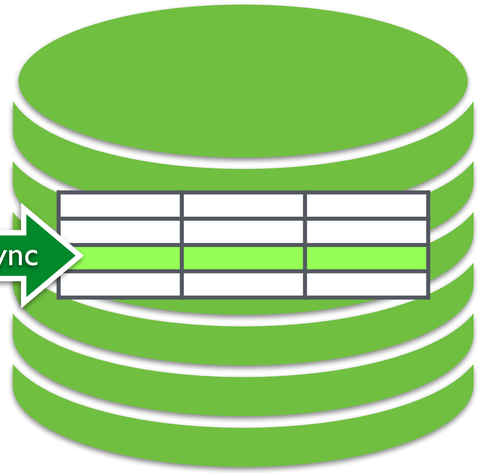
no easy navigation and querying of the serialized object graph

no support of legacy persistent data, stored in relational databases

mapping objects to relational databases







relational database

payara COMMUNITY

JAKARTA EE

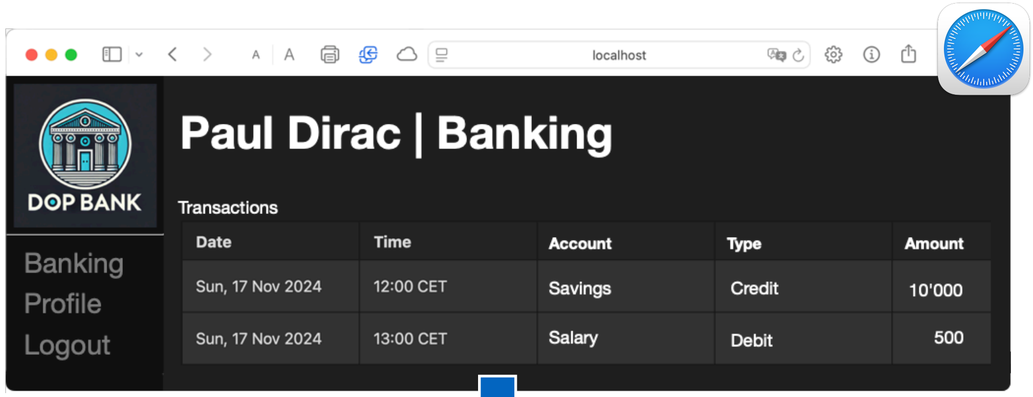
application server

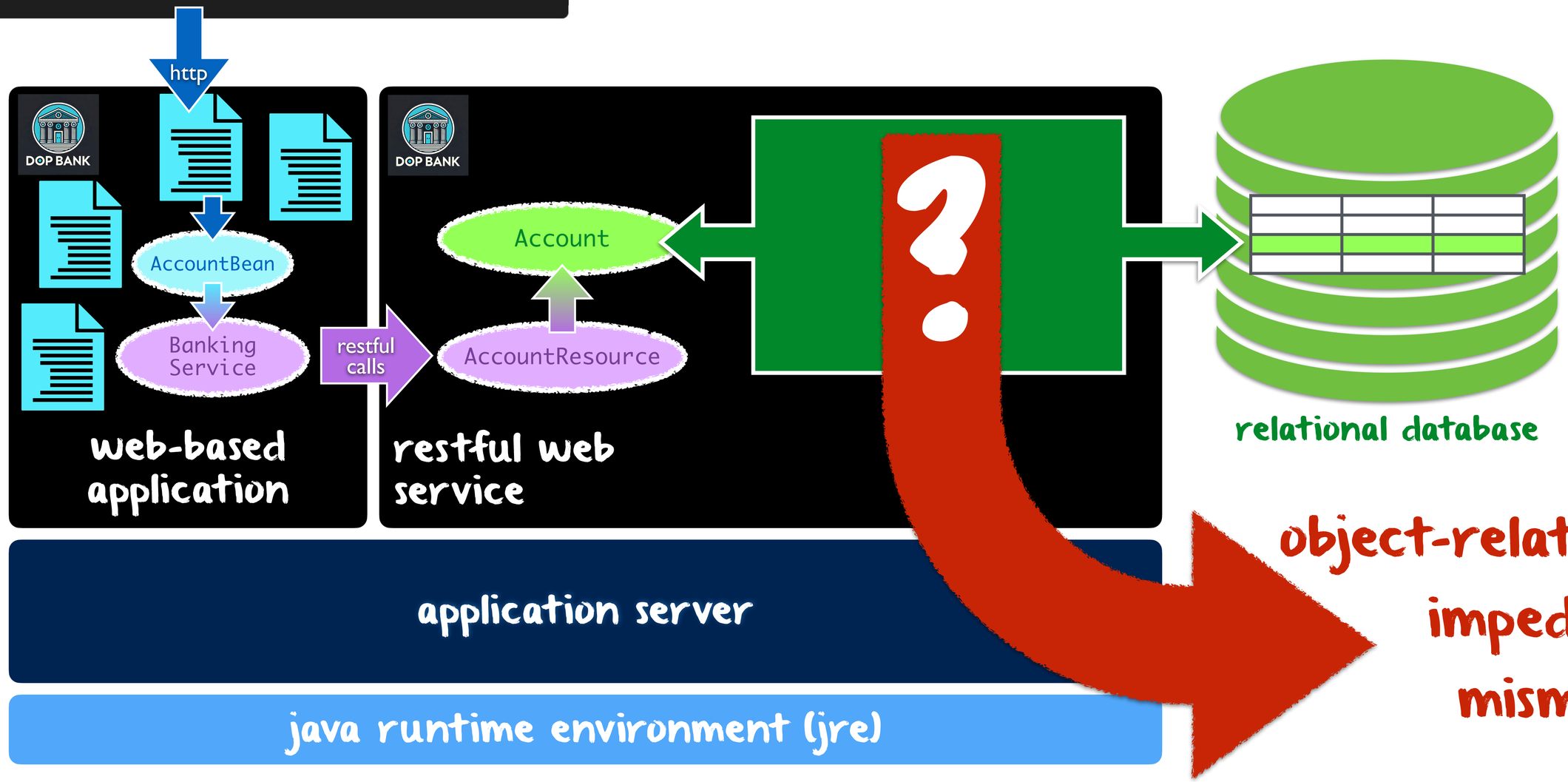
Java

java runtime environment (jre)

keeping in-memory objects synchronized with data persisted in a relational database is object-relational mapping

mapping objects to relational databases





object-relational impedance mismatch

😕 **object identity**

- an object is uniquely identified by its implicit memory address
- a row is uniquely identified by its explicit primary key

😕 **object relations**

- an object references other objects by directly or indirectly holding references to them, i.e., by holding their implicit memory addresses
- a row references other rows via explicit foreign keys stored somewhere in the database, i.e., in the same table or in a join table

😕 **object methods**

- an object does not only encapsulate state (data), it also offers operations (methods) to manipulate that state
- a row contains data only, i.e., it comes with no set of operations to safely manipulate that data or do complex computations with it

😕 **class inheritance**

- the object model allows a class to inherit attributes from another class
- the relational model offers no way to express inheritance between tables

object-relational impedance mismatch

😕 object identity

😕 object methods

😕 object relations

😕 class inheritance

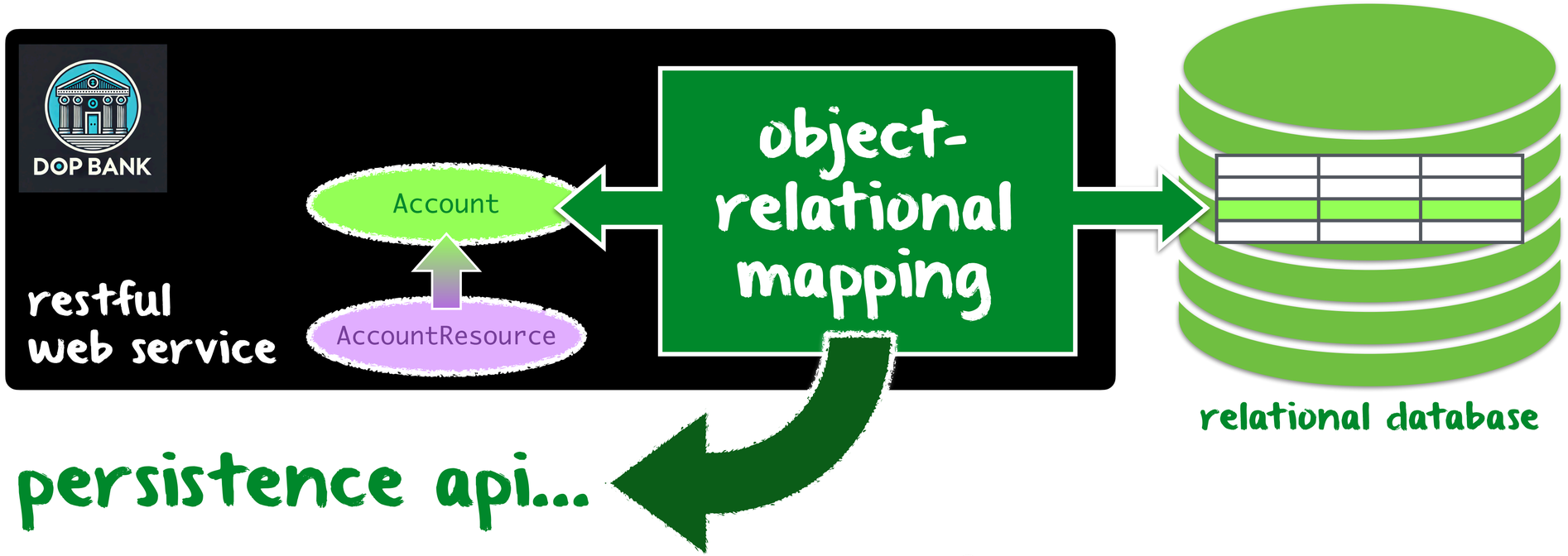
🤔 which one should be the master model ?

the relational model ?

the object model ?

who should be responsible for the mapping ? 🤔

who should be responsible for the mapping ?



the java persistence api...

... proposes declarative support for object-relational mapping

... is available for all java editions and container models

... is independent of operating systems and databases

... is based on the notion of entity

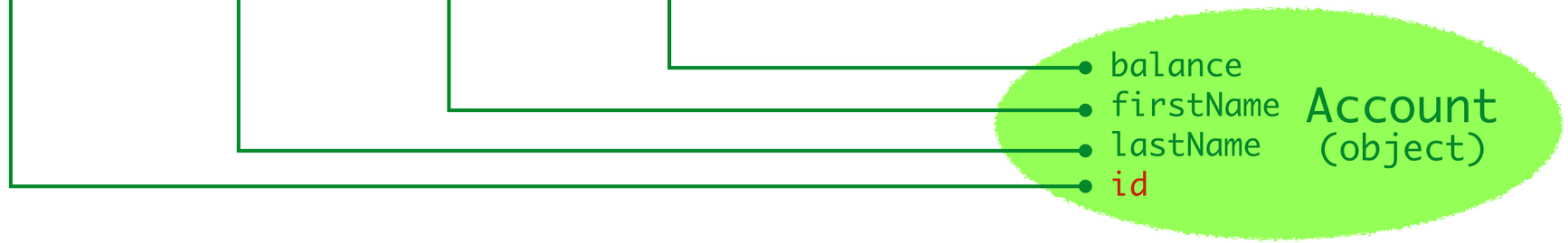
what is an entity ?

- an entity is a java object representing data from a relational database
- an entity's lifetime is independent of the application lifetime using it

Account (table)

id	lastName	firstName	balance
121799	"Simpson"	"Marge"	2000

- an entity also has a persistent identity, i.e., its primary key, that is distinct from its object reference in memory



object identity ➧ in volatile memory, an entity is identified by its address

➧ in the database, an entity is identified by its primary key

what is an entity ?

```java
@Entity
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "BALANCE")
    private double balance = 0.0;

    public Account(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    ⋮
    public Account() {
        this.lastName = null;
        this.firstName = null;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "-" + String.format("%08X", id)
                + "[" + lastName + ", " + firstName + ", $" + balance + "]";
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" … >
  <persistence-unit name="PU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ch.unil.doplab.persistence.Account</class>
    <class>ch.unil.doplab.persistence.Payment</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp:localhost:9092/~/databases/myh2"/>
      <property name="javax.persistence.jdbc.user" value="sa"/>
      <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

```java
public static void persist(Object object) {
  EntityManagerFactory factory = Persistence.createEntityManagerFactory("PU");
  EntityManager manager = factory.createEntityManager();
  manager.getTransaction().begin();
  try {
    manager.persist(object);
    manager.getTransaction().commit();
  } catch (Exception e) {
    e.printStackTrace();
    manager.getTransaction().rollback();
  } finally {
    manager.close();
  }
}
```

```java
public class Main {
  public static void main(String args[]) {
       Account margeAccount = new Account("Simpson", "Marge");
       margeAccount.setBalance(2000.0);
       System.out.println(margeAccount);
       persist(margeAccount);
       System.out.println(margeAccount);
   }
}
```

```
Account-00000000[Simpson, Marge, $2000.0]
```

```
Account-00000001[Simpson, Marge, $2000.0]
```

account

id	lastName	firstName	balance
1	"Simpson"	"Marge"	2000

```java
public static Account findAccountByPrimaryKey(Long id) {
  Account account = null;
  EntityManagerFactory factory = Persistence.createEntityManagerFactory("PU");
  EntityManager manager = factory.createEntityManager();
  manager.getTransaction().begin();
  try {
    account = manager.find(Account.class, id);
    manager.getTransaction().commit();
  } catch (Exception e) {
    e.printStackTrace();
    manager.getTransaction().rollback();
  } finally {
    manager.close();
  }
  return account;
}
```

what is an entity ?

entity lookup & queries

in addition to the find-by-primary-key query, we can perform more general queries to find entities, using the Java Persistence Query Language (JP-QL)

JP-QL is similar to SQL but it manipulates objects and fields rather than rows & columns

queries can be dynamic or static

static queries are called named queries

dynamic query

```java
public List<Account> findAllAccounts() {
  Query query = manager.createQuery("SELECT a FROM Account a");
  return query.getResultList();
}
```

static query

```java
public List<Account> findAccountsByLastName(String lastName) {
  Query query = manager.createNamedQuery("findByLastName");
  return query.setParameter("lastName", lastName).getResultList();
}
```

```java
@Entity
@NamedQueries({
        @NamedQuery(name = "findByAccountID",  query = "SELECT a FROM Account a WHERE a.id = :id"),
        @NamedQuery(name = "findByLastName",   query = "SELECT a FROM Account a WHERE a.lastName = :lastName"),
        @NamedQuery(name = "findByBalance",    query = "SELECT a FROM Account a WHERE a.balance = :balance")})
public class Account implements Serializable {
⋮
}
```

what is an entity ?

object relations ➡ an object references other objects by directly or indirectly holding references to them, i.e., by holding their implicit memory addresses

➡ a row references other rows via explicit foreign keys stored somewhere in the database, i.e., in the same table or in a join table

```java
@Entity
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "BALANCE")
    private double balance = 0.0;

    @OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private List<Payment> payments;

    public void addPayment(Payment payment) {
        payment.setAccount(this);
        payments.add(payment);
    }
    ⋮
}
```

```java
@Entity
public class Payment implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "EXECDATE")
    private Date execDate;

    @Column(name = "AMOUNT")
    private double amount;

    @ManyToOne
    private Account account;
    ⋮
}
```

account

id	lastName	firstName	balance
1	"Simpson"	"Marge"	2000

payment

id	amount	execdate	account_id
4	150.0	2019-01-02	1
5	300.0	2019-02-04	1
6	450.0	2019-03-06	1
7	600.0	2019-04-08	1

what is an entity ?

object relations ➡ an object references other objects by directly or indirectly holding references to them, i.e., by holding their implicit memory addresses

➡ a row references other rows via explicit foreign keys stored somewhere in the database, i.e., in the same table or in a join table

```java
@Entity
public class Account implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "BALANCE")
    private double balance = 0.0;

    @OneToMany(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private List<Payment> payments;

    public void addPayment(Payment payment) {
        payment.setAccount(this);
        payments.add(payment);
    }
    ⋮
}
```

```java
@Entity
public class Payment implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "EXECDATE")
    private Date execDate;

    @Column(name = "AMOUNT")
    private double amount;

    @ManyToOne
    private Account account;
    ⋮
}
```

account

id	lastName	firstName	balance
1	"Simpson"	"Marge"	2000

payment

id	amount	execdate
4	150.0	2019-01-02
5	300.0	2019-02-04
6	450.0	2019-03-06
7	600.0	2019-04-08

account_payment

account_id	payment_id
1	4
1	5
1	6
1	7

object-relational impedance mismatch

who should be responsible for the mapping ? → Java Persistence API (JPA)

object identity

- in volatile memory, an entity is identified by its address
- in the database, an entity is identified by its primary key

object relations

- in volatile memory, an entity references other entities by directly or indirectly holding references to them, as for any java object
- in the database, an entity references other entities via explicit foreign keys stored in the same table and/or in a separate join table

object identity and relations are expressed and managed automatically and transparently by the Java Persistence API (JPA)

object-relational impedance mismatch

object methods ➡ an object does not only encapsulate state (data), it also offers operations (methods) to manipulate that state

➡ a row contains data only, i.e., it comes with no set of operations to properly manipulate that data or do complex computations with it

➡ **deep mismatch ➡ no mapping possible**

---

class inheritance ➡ the object model allows a class to inherit attributes from another class

➡ the relational model offers no way to express inheritance between tables

➡ **four different approaches are proposed:**

- mapped superclass : simple class (no entity, no table) defining common fields for multiple entities*
- one table per entity hierarchy : all fields of all entity subclasses are mapped to a single table
- one table per entity : each entity class has its own table and all its fields are mapped to it
- joined entity tables : only the specific fields of each entity class is mapped to its table

⇒ the persistent manager must perform table joins to persist each entity

* in addition, no polymorphic queries

mapped superclass

Medium

`title`
`rating`

`init`
`rate`
`printInfo`

Book

`author`

`init`

Vinyl

`artist`
`duration`

`init`

Book (table)

id	title	rating	author
121799	"Hamlet"	4	"Shakespeare"
...	...	...	...

Vinyl (table)

id	title	rating	artist	duration
294875	"Synchronicity"	3	"The Police"	40
...	...	...	...	...

mapped superclass

```java
@MappedSuperclass
public abstract class Medium implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "TITLE")
    private String title;

    @Column(name = "RATING")
    private int rating;
    ⋮
}
```

```java
@Entity
public class Book extends Medium {

    @Column(name = "AUTHOR")
    private String author;
    ⋮
}
```

```java
@Entity
public class Vinyl extends Medium {

    @Column(name = "ARTIST")
    private String artist;

    @Column(name = "DURATION")
    private int duration;
    ⋮
}
```

the superclass is a purely design abstraction, i.e., it is not an entity, nor is it mapped to a database table

the superclass is often abstract

no support of polymorphic queries

joined entity tables

Medium (table)

id	title	rating
804744	"Hamlet"	4
982075	"Synchronicity"	3
…	…	…

Medium

title
rating

init
rate
printInfo

Book

author

init

Vinyl

artist
duration

init

Book (table)

id	author	medium
121799	"Shakespeare"	804744
…	…	…

Vinyl (table)

id	artist	duration	medium
294875	"The Police"	40	982075
…	…	…	…

joined entity tables

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Medium implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id = 0L;

    @Column(name = "TITLE")
    private String title;

    @Column(name = "RATING")
    private int rating;
    ⋮
}
```

```java
@Entity
public class Book extends Medium {

    @Column(name = "AUTHOR")
    private String author;
    ⋮
}
```

```java
@Entity
public class Vinyl extends Medium {

    @Column(name = "ARTIST")
    private String artist;

    @Column(name = "DURATION")
    private int duration;
    ⋮
}
```

structural similarity between classes and their corresponding relational tables

support of polymorphic queries

entity lifecycle callbacks

```
@Entity
@Table(name = "ACCOUNT")
public class Account {
    @PrePersist
    void prePersist() { ... }

    @PostPersist
    void postPersist()  { ... }

    @PreRemove
    void preRemove()  { ... }

     ⋮

     ⋮

    @PostRemove
    void postRemove() { ... }

    @PreUpdate
    void preUpdate()  { ... }

    @PostUpdate
    void postUpdate() { ... }

    @PostLoad
    void postLoad() { ... }
}
```

using entities from cdi beans

```
@ApplicationScoped
public class ApplicationState {
    @PersistenceContext(unitName = "BankPU"))
    private EntityManager manager;

    private Account account = null;

    @Transactional
    public void open(int accountNumber) {
        account = manager.find(Account.class, accountNumber);
        if (account == null) {
            account = new Account();
            manager.persist(account);
        }
    }
    @Transactional
    public void deposit(int amount) {
        if (account == null) throw new IllegalStateException();
        account = manager.merge(account)
        account.deposit(amount);
    }
   public String getLastName() {
        if (account == null) throw new IllegalStateException();
        return account.getLastName();
    }
    ⋮
}
```

dependency injection

entity states & transactions

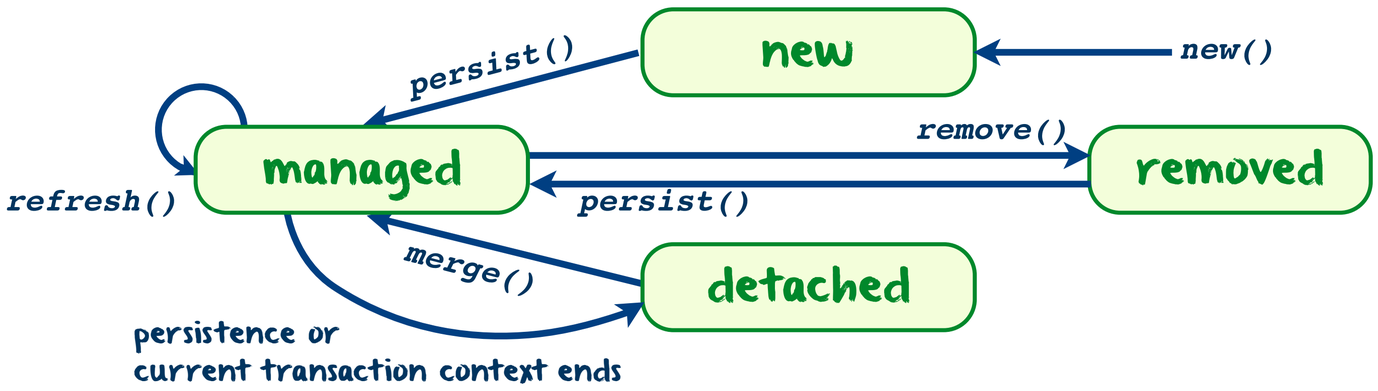
after a `manager.persist(account)` or a `manager.merge(account)` call, the account entity is scheduled for being synchronized (written) to the database

the entity will actually be written when the current transaction commits

until then, the entity is in managed state

an entity is only kept in sync with the database when in managed state

an entity is only in managed state when manipulated in the context of a transaction



new	just created but not yet bound to a persistent identity in the database
managed	bound to a persistent identity and scheduled to be synchronized with the database
detached	bound to a persistent identity but not synchronized with the database
removed	bound to a persistent context and scheduled for removal from the database

atomic transactions

an atomic transaction (often simply transaction),
is an indivisible sequence of database operations
such that either all occur, or none occurs

a transaction is local if it involves only one database
and global if it involves two or more databases

more formally, this implies that
a transaction ensures the four acid properties

atomic transactions

a transaction T ensures the four acid properties:

- **a**tomicity — T appears either committed or aborted with respect to failures
- **c**onsistency — T does not compromise the consistency of the data it manipulates
- **i**solation — T appears indivisible with respect to all other transactions
- **d**urability — T being committed, its effects will survive subsequent crashes

this is ensured by persistence

using entities from cdi beans

with transactions

```
@ApplicationScoped
public class ApplicationState {
    @PersistenceContext(unitName = "BankPU"))
    private EntityManager manager;

    private Account account = null;

    @Transactional
    public void open(int accountNumber) {
        account = manager.find(Account.class, accountNumber);
        if (account == null) {
            account = new Account();
            manager.persist(account);
        }
    }
    @Transactional
    public void deposit(int amount) {
        if (account == null) throw new IllegalStateException();
        account = manager.merge(account)
        account.deposit(amount);
    }
   public String getLastName() {
        if (account == null) throw new IllegalStateException();
        return account.getLastName();
    }
    ⋮
}
```

```
@Transactional
@ApplicationScoped
public class ApplicationState {
    @PersistenceContext(unitName = "BankPU"))
    private EntityManager manager;

    private Account account = null;


    public void open(int accountNumber) {
        account = manager.find(Account.class, accountNumber);
        if (account == null) {
            account = new Account();
            manager.persist(account);
        }
    }

    public void deposit(int amount) {
        if (account == null) throw new IllegalStateException();
        account = manager.merge(account)
        account.deposit(amount);
    }
   public String getLastName() {
        if (account == null) throw new IllegalStateException();
        return account.getLastName();
    }
    ⋮
}
```

transactional types

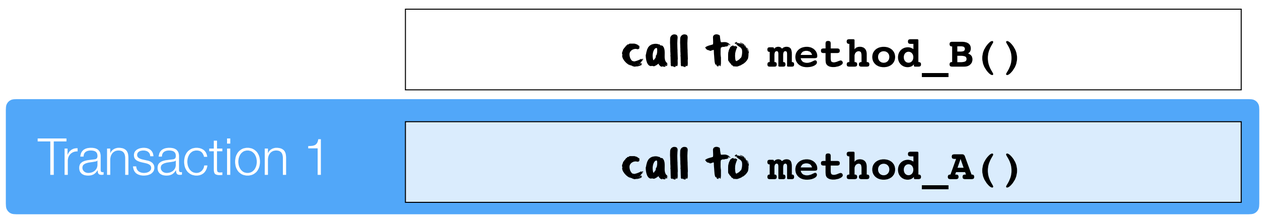
a transactional type controls whether a bean method is to be executed within a transaction context

type	meaning
`NotSupported`	if a client's transaction exists, it is suspended
`Supports`	if a client's transaction exists, it is continued
`Required`	if a client's transaction exists, it is continued, otherwise, the container starts a new transaction
`RequiresNew`	the container always starts a new transactions, if a client's transaction exists, it is suspended first
`Mandatory`	the client must be in a transaction when calling
`Never`	the client must not be in a transaction when calling

transactional types

```
@ApplicationScoped
@Transactional(TxType.REQUIRED)
public class ApplicationState {
     ⋮
     @Transactional(TxType.REQUIRES_NEW)
     public void open(int accountNumber) { ... }
}
```

call stack



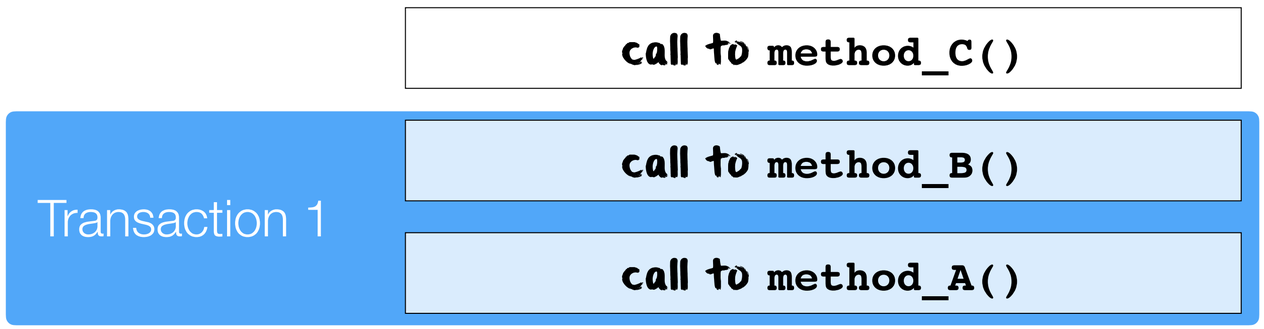
transactional types

method_G()	**Mandatory**
method_F()	**Required**
method_E()	**Supports**
method_D()	**NotSupported**
method_C()	**RequiresNew**
method_B()	**Supports**
method_A()	**Required**

transactional types

```
@ApplicationScoped
@Transactional(TxType.REQUIRED)
public class ApplicationState {
     ⋮
     @Transactional(TxType.REQUIRES_NEW)
     public void open(int accountNumber) { ... }
}
```

call stack



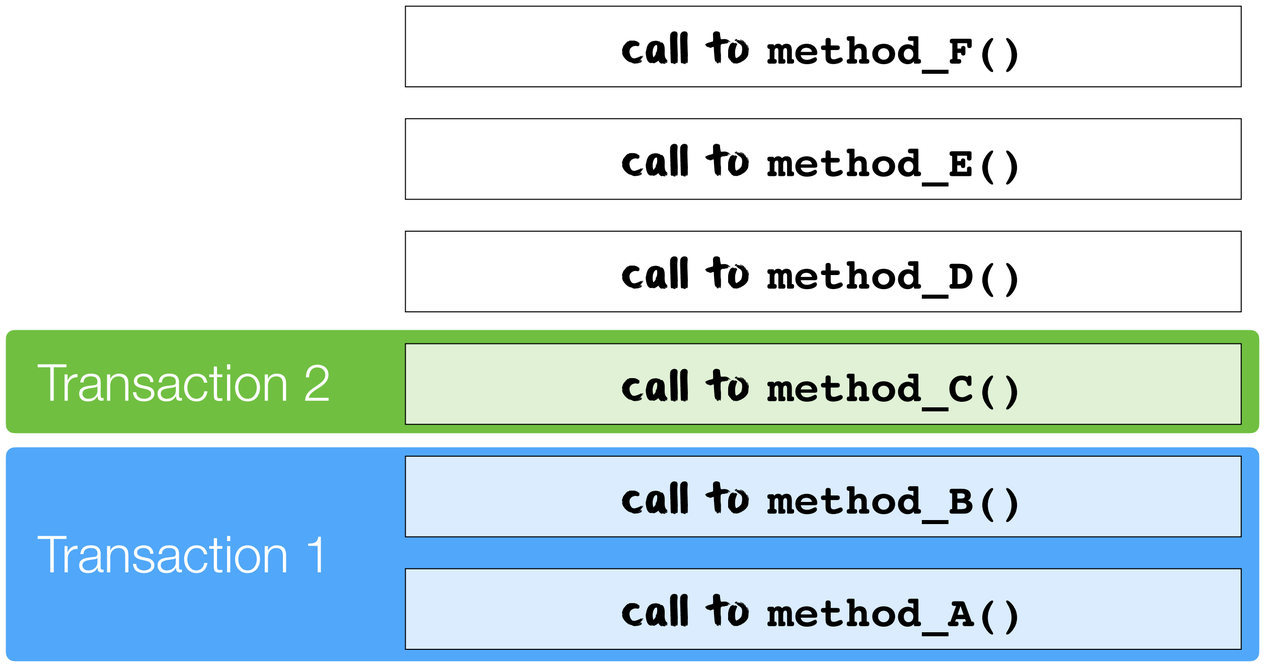
transactional types

method_G()	**Mandatory**
method_F()	**Required**
method_E()	**Supports**
method_D()	**NotSupported**
method_C()	**RequiresNew**
method_B()	**Supports**
method_A()	**Required**

transactional types

```
@ApplicationScoped
@Transactional(TxType.REQUIRED)
public class ApplicationState {
     ⋮
     @Transactional(TxType.REQUIRES_NEW)
     public void open(int accountNumber) { ... }
}
```

call stack

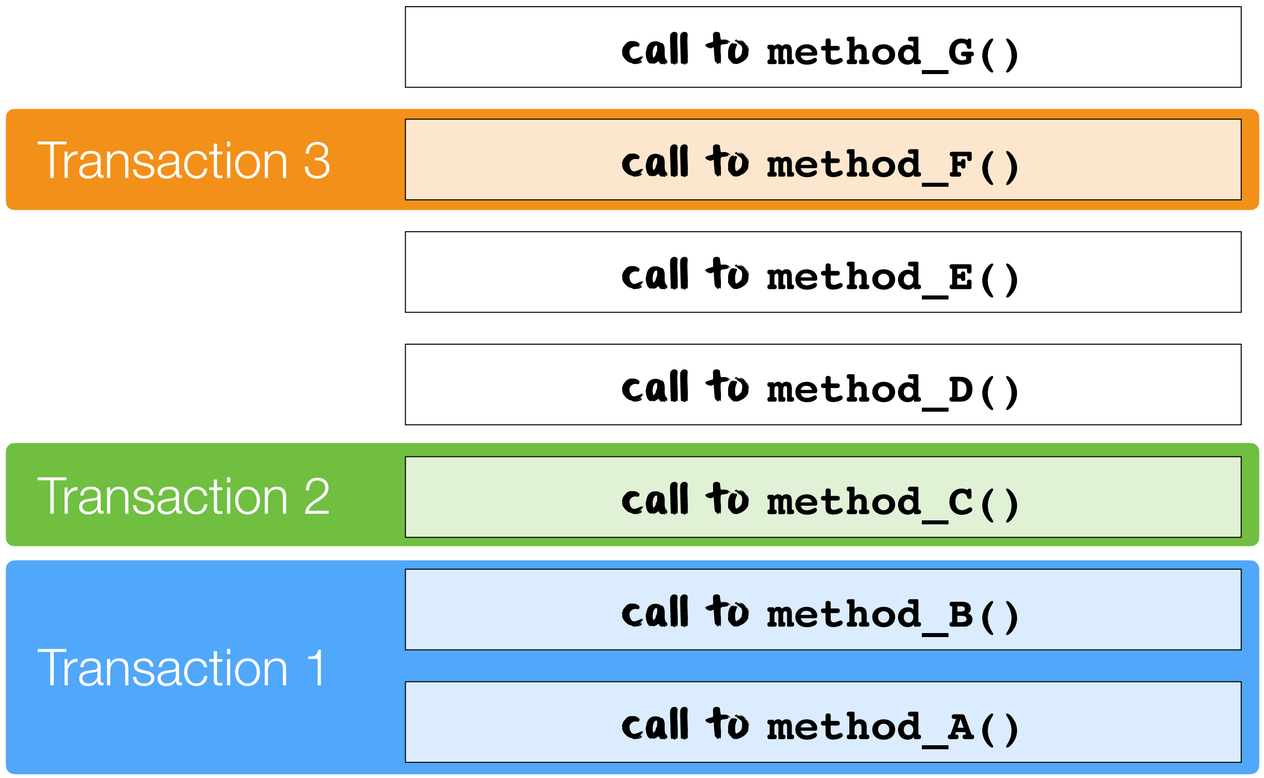


transactional types

method_G()	**Mandatory**
method_F()	**Required**
method_E()	**Supports**
method_D()	**NotSupported**
method_C()	**RequiresNew**
method_B()	**Supports**
method_A()	**Required**

transactional types

```
@ApplicationScoped
@Transactional(TxType.REQUIRED)
public class ApplicationState {
     ⋮
     @Transactional(TxType.REQUIRES_NEW)
     public void open(int accountNumber) { ... }
}
```



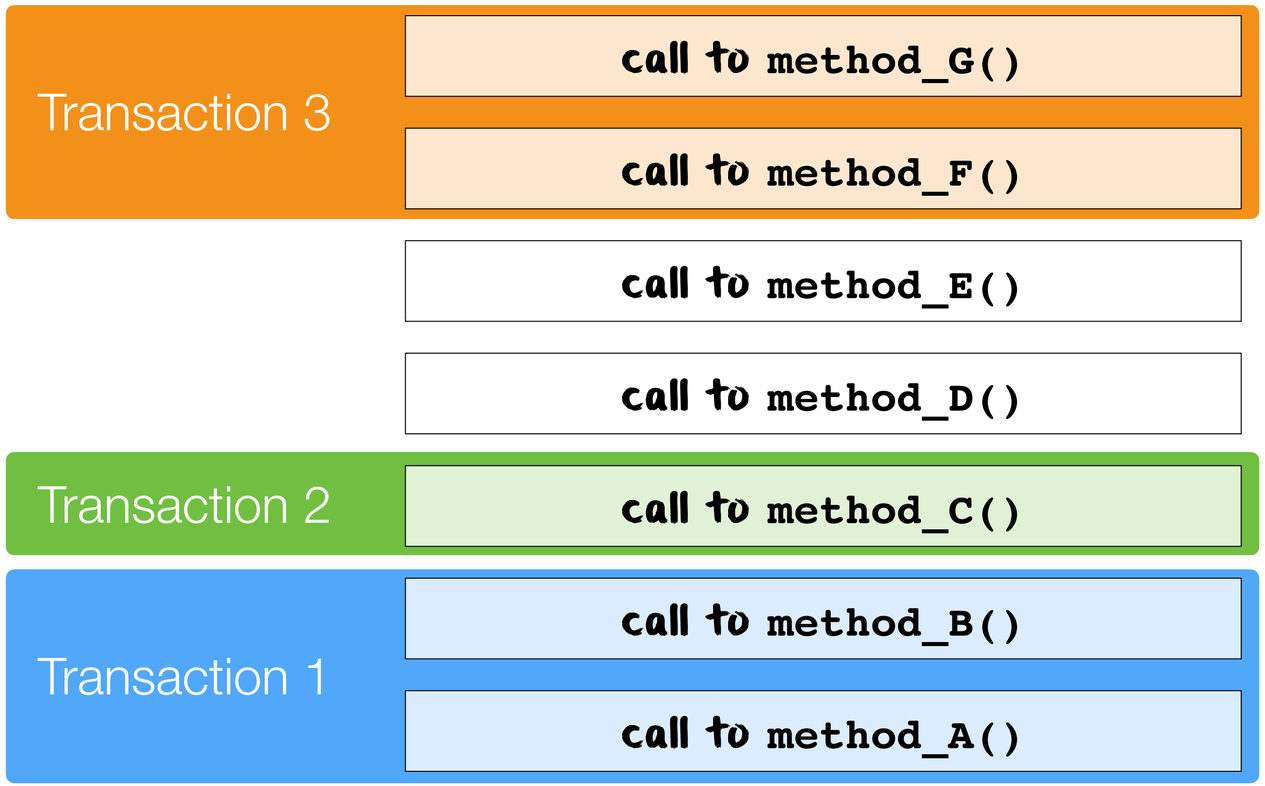
transactional types

method_G()	**Mandatory**
method_F()	**Required**
method_E()	**Supports**
method_D()	**NotSupported**
method_C()	**RequiresNew**
method_B()	**Supports**
method_A()	**Required**

transactional types

```
@ApplicationScoped
@Transactional(TxType.REQUIRED)
public class ApplicationState {
     ⋮
     @Transactional(TxType.REQUIRES_NEW)
     public void open(int accountNumber) { ... }
}
```

call stack



transactional types

method_G()	**Mandatory**
method_F()	**Required**
method_E()	**Supports**
method_D()	**NotSupported**
method_C()	**RequiresNew**
method_B()	**Supports**
method_A()	**Required**

transactional types

how to tell the container to rollback a transaction, because of some applicative problem occurred?

```
@ApplicationScoped
public class ApplicationState {
    @Resource
    TransactionSynchronizationRegistry registry;
    ⋮
    @Transactional
    public void withdraw(int amount) throws InsufficientBalanceException{
        if (account.getBalance() - amount < 0) {
            registry.setRollbackOnly();
            throw new InsufficientBalanceException();
        }
        account.withdraw(amount);
    }
    ⋮
}
```

dependency injection