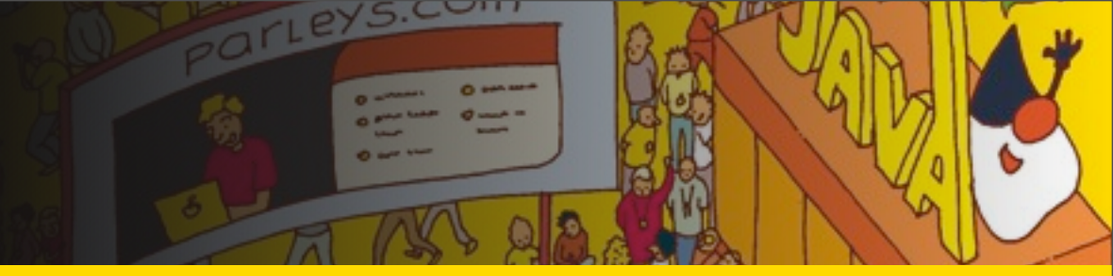
Performance Anti-Patterns
in Hibernate

Patrycja Wegrzynowicz
CTO
Yonita, Inc.

About Me



- 10+ years of professional experience
- PhD in Computer Science
  - Patterns and anti-patterns, static and dynamic code analysis, language semantics, compiler design
- Active author and speaker
  - JavaOne, Devoxx, JavaZone, Jazoon, OOPSLA, ASE...
- CTO of Yonita, Inc.
  - Bridge the gap between the industry and the academia
  - Automated defects detection and optimization
  - Performance, security, concurrency, databases
- Twitter @yonlabs

Agenda

- 4 puzzles
  - –Short program with curious behavior
  - –Question to you (multiple choice)
  - –Mystery revealed
  - –Lessons learned
- 7 performance anti-patterns
  - –Description, consequences, solutions

Disclaimer

I do think Hibernate is a great tool!

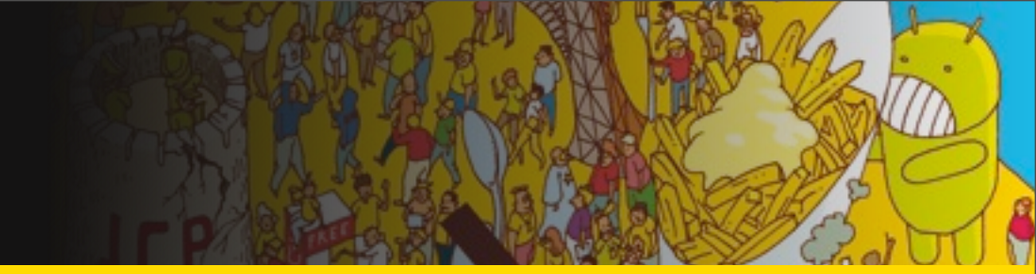
Puzzle #1

Less Is More

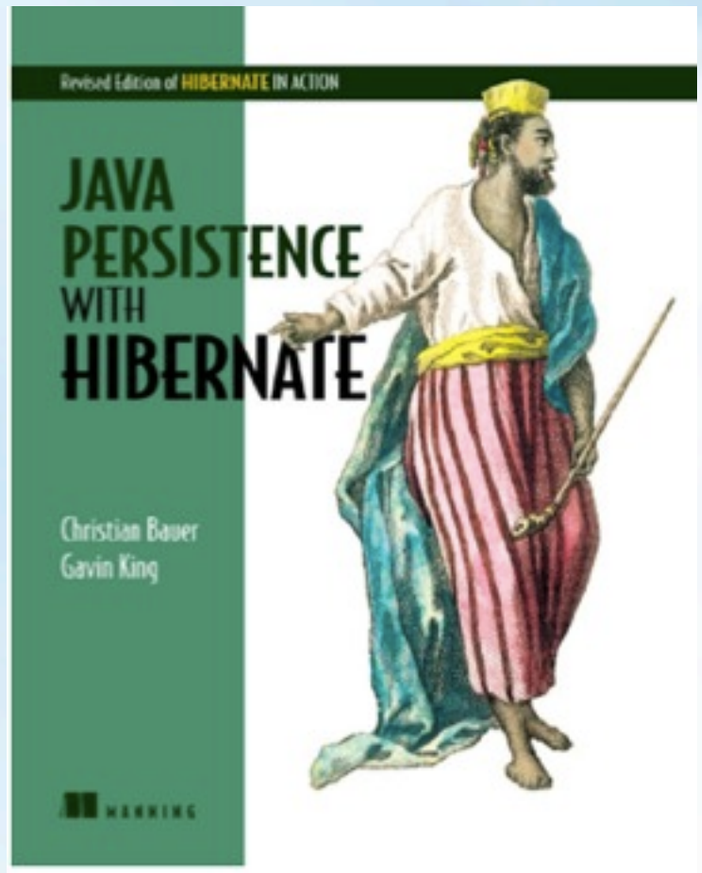
Who knows CaveatEmptor app?

Who's read Java Persistence with Hibernate?

CaveatEmptor



- Demo application from ‘Java Persistence with Hibernate’
- Simple auction system



I’m Gonna Win This Auction!

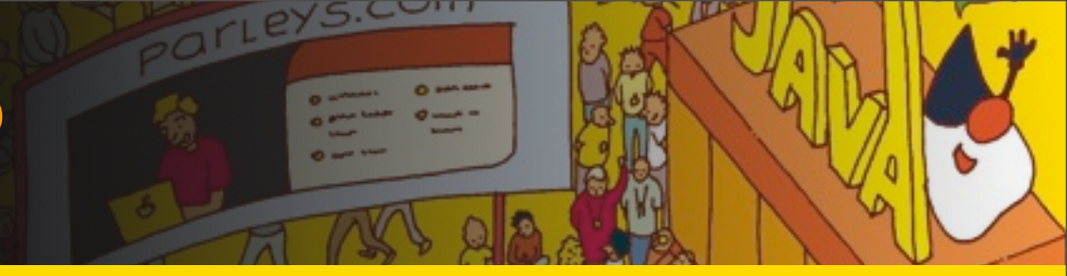
```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

What's the Problem?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

(a) None (code is correct)
(b) Transaction/lock mgmt
(c) Exception thrown
(d) None of the above

What's the Problem?



(a) None (the code is correct)

(b) Transaction/Lock management

(c) Exception thrown

(d) None of the above

The bids can change between calling getMaxBid and locking the item.

Another Look

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);
        // here, the bids can be modified by a different transaction
        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

2 Users and Item with 2 Bids

Thread A (new amount 100)

(1)

```
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20
```

Thread B (new amount 30)

2 Users and Item with 2 Bids

Thread A (new amount 100)

(1)

```
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20
```

Thread B (new amount 30)

(2)

```
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20
```

2 Users and Item with 2 Bids

Thread A (new amount 100)

```
                (1)
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20

                (3)
// item with 2 bids: 10,20
// Lock.UPGRADE
Item item = dao.findById(id,true);
Bid newBid = item.placeBid(...,
             newAmount,
             curMax, ...);
// commit transaction
// item with 3 bids: 10,20,100
```

Successful bid: 100

Thread B (new amount 30)

```
                (2)
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20
```

2 Users and Item with 2 Bids

Thread A (new amount 100)

```
                (1)
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20

                (3)
// item with 2 bids: 10,20
// Lock.UPGRADE
Item item = dao.findById(id,true);
Bid newBid = item.placeBid(...,
             newAmount,
             curMax, ...);
// commit transaction
// item with 3 bids: 10,20,100
```

Successful bid: 100

Thread B (new amount 30)

```
                (2)
// begin transaction
// item with 2 bids: 10,20
Bid curMax = dao.getMaxBid(id);
// curMax = 20

                (4)
// item with 3 bids: 10,20,100
// Lock.UPGRADE
Item item = dao.findById(id,true);
Bid newBid = item.placeBid(...,
             newAmount,
             curMax, ...);
// commit transaction
// item with 4 bids: 10,20,100,30
```

Successful bid: 30

How To Fix It?

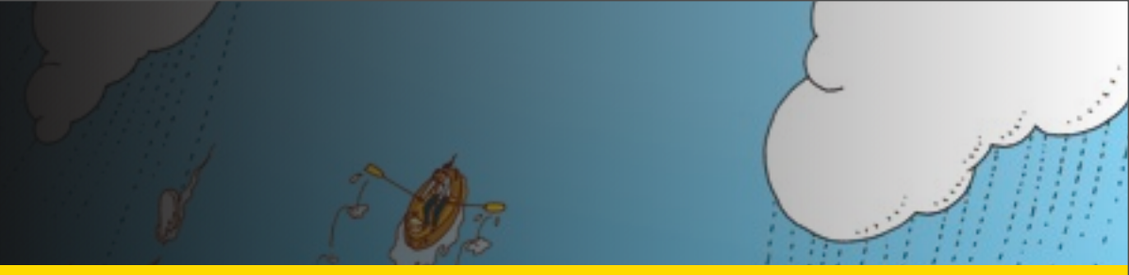
```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

How To Fix It?

version and optimistic locking

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

How To Fix It?



version and optimistic locking **NO**

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

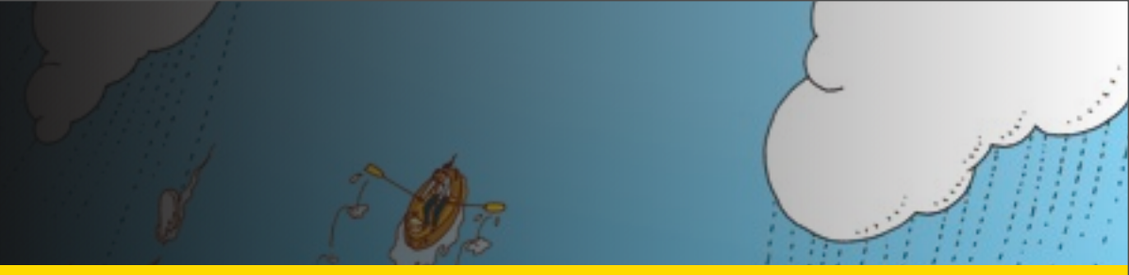
How To Fix It?

version and optimistic locking NO

REPEATABLE READS

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

How To Fix It?



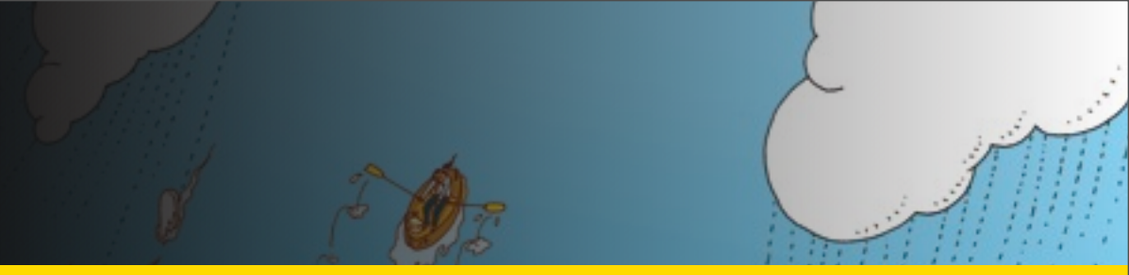
version and optimistic locking NO

REPEATABLE READS

SERIALIZABLE

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

How To Fix It?



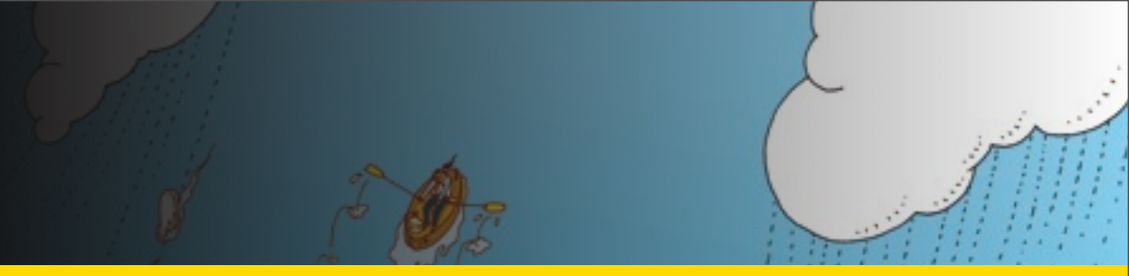
version and optimistic locking — NO

REPEATABLE READS — NO

SERIALIZABLE

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

How To Fix It?



version and optimistic locking — NO

REPEATABLE READS — NO

SERIALIZABLE — YES

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

How To Fix It?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

version and optimistic locking — NO

REPEATABLE READS — NO

SERIALIZABLE — YES

move lock before getMinBid/ getMaxBid

How To Fix It?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

version and optimistic locking — NO

REPEATABLE READS — NO

SERIALIZABLE — YES

move lock before getMaxBid/ getMaxBid — YES

How To Fix It?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

version and optimistic locking — NO

REPEATABLE READS — NO

SERIALIZABLE — YES

move lock before getMaxBid/ getMaxBid — YES

different object model

How To Fix It?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

version and optimistic locking — NO

REPEATABLE READS — NO

SERIALIZABLE — YES

move lock before getMinBid/ getMaxBid — YES

different object model — YES

How To Fix It?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

- version and optimistic locking — NO
- REPEATABLE READS — NO
- SERIALIZABLE — YES
- move lock before getMaxBid/getMinBid — YES
- different object model — YES
- Java synchronized

How To Fix It?

```
public class PlaceBidAction implements Action {
    // invoked in a new transaction
    public String execute(Map even) {
        // ...
        Bid currentMinBid = itemDAO.getMinBid(itemId);
        Bid currentMaxBid = itemDAO.getMaxBid(itemId);

        // uses load with Lock.UPGRADE
        Item item = itemDAO.findById(itemId, true);
        Bid newBid = item.placeBid(userDAO.findById(userId, false),
                                   newAmount,
                                   currentMaxBid,
                                   currentMinBid);
        // ...
    }
}
```

version and optimistic locking — NO

REPEATABLE READS — NO

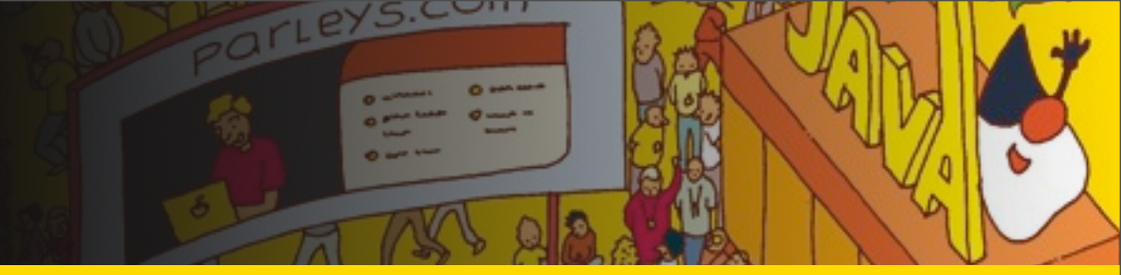
SERIALIZABLE — YES

move lock before getMinBid/ getMaxBid — YES

different object model — YES

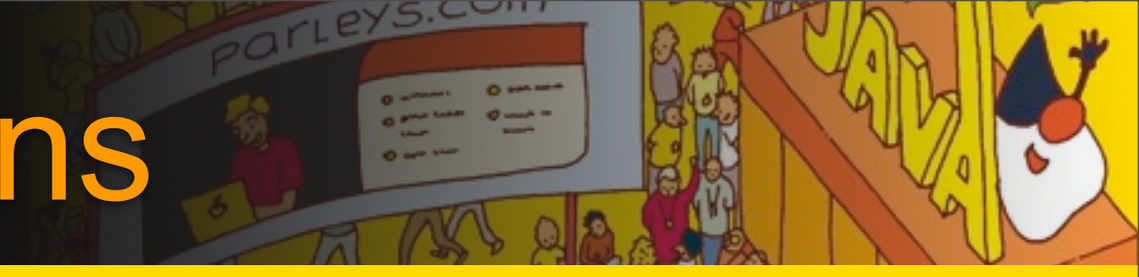
synchronized — YES

Lessons Learned



- Nobody's perfect ;-)
- ACID is only a theory
  - –Simple transaction interceptor/annotation often is not enough
  - –Query order does matter
  - –Learn more about locks and transaction isolation levels
- Premature optimization is the root of all evil
  - –Performance Anti-Pattern #1

Performance Lessons



- Object model
- Usage patterns
  - Concurrent users
  - Data volume
- Transaction and locking management

Puzzle #2

Heads of Hydra

Heads of Hydra

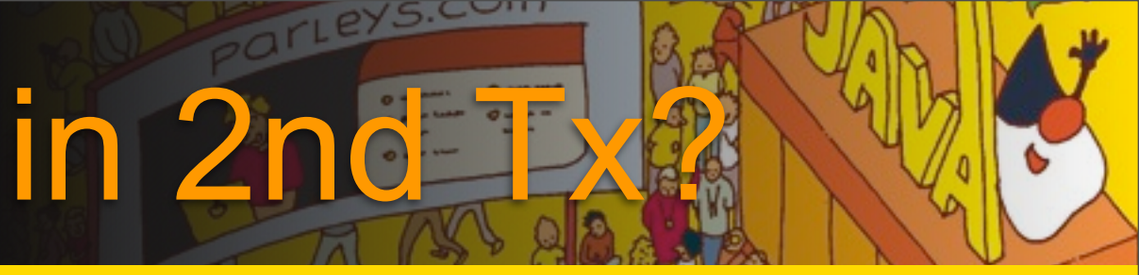
```
@Entity
public class Hydra {
    private Long id;
    private List<Head> heads = new ArrayList<Head>();

    @Id @GeneratedValue
    public Long getId() {...}
    protected void setId() {...}
    @OneToMany(cascade=CascadeType.ALL)
    public List<Head> getHeads() {
        return Collections.unmodifiableList(heads);
    }
    protected void setHeads(List<Head> heads) {...}
}

// creates and persists the hydra with 3 heads

// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

How Many Queries in 2nd Tx?

```
@Entity
public class Hydra {
    private Long id;
    private List<Head> heads = new ArrayList<Head>();

    @Id @GeneratedValue
    public Long getId() {...}
    protected void setId() {...}
    @OneToMany(cascade=CascadeType.ALL)
    public List<Head> getHeads() {
        return Collections.unmodifiableList(heads);
    }
    protected void setHeads(List<Head> heads) {...}
}

// creates and persists the hydra with 3 heads

// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

(a) 1 select
(b) 2 selects
(c) 1+3 selects
(d) 2 selects, 1 delete, 3 inserts
(e) None of the above

How Many Queries in 2nd Tx?



(a) 1 select

(b) 2 selects

(c) 1+3 selects

(d) 2 selects, 1 delete, 3 inserts

(e) None of the above

During commit hibernate checks whether the collection property is dirty (needs to be re-created) by comparing Java identities (object references).

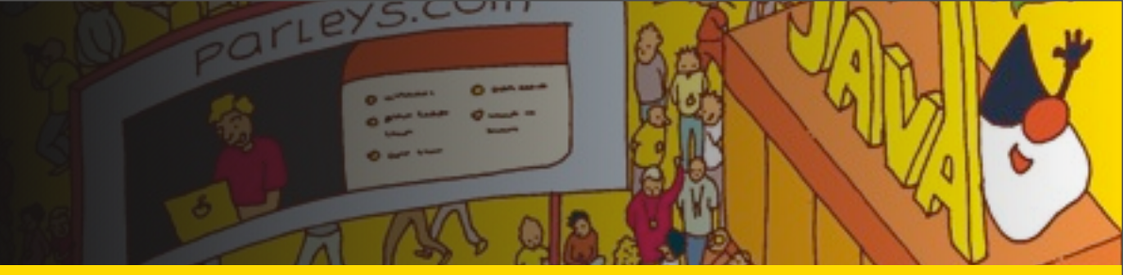
Another Look

```
@Entity
public class Hydra {
    private Long id;
    private List<Head> heads = new ArrayList<Head>();

    @Id @GeneratedValue
    public Long getId() {...}
    protected void setId() {...}
    @OneToMany(cascade=CascadeType.ALL)
    public List<Head> getHeads() {
        return Collections.unmodifiableList(heads);
    }
    protected void setHeads(List<Head> heads) {...}
}

// creates and persists hydra with three heads

// new EntityManager and new transaction
// during find only 1 select (hydra)
Hydra found = em.find(Hydra.class, hydra.getId());
// during commit 1 select (heads),1 delete (heads),3 inserts (heads)
```

How To Fix It?

```
@Entity
public class Hydra {
    @Id @GeneratedValue
    private Long id;
    @OneToMany(cascade=CascadeType.ALL)
    private List<Head> heads = new ArrayList<Head>();

    public Long getId() {...}
    protected void setId() {...}
    public List<Head> getHeads() {
        return Collections.unmodifiableList(heads);
    }
    protected void setHeads(List<Head> heads) {...}
}

// creates and persists hydra with three heads

// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

Lessons Learned



- Expect unexpected ;-)
- Prefer field access mappings
- And...

Anti-Pattern #2

Lost Collection Proxy on Owning Side

Lost Collection Proxy

- Description
  - On the owning side of an association, we assign a new collection object to a persistent field/property overriding a collection proxy returned by hibernate
- Consequences
  - The entire collection is re-created: **one delete** to remove all elements and then **many element inserts**

Lost Collection Proxy - Example

```
@Entity
public class Hydra {
    @Id @GeneratedValue
    private Long id;
    @OneToMany(cascade=CascadeType.ALL)
    private List<Head> heads = new ArrayList<Head>();

    public Long getId() {...}
    protected void setId() {...}
    public List<Head> getHeads() {
        return heads;
    }
    public void setHeads(List<Head> heads) {
        this.heads = heads;
    }
}
```

Lost Collection Proxy - Solution

- Operate on collection objects returned by hibernate
  - –In most cases it's a much more efficient solution
- Unless you know what you're doing
  - –It might be performance-wise in some cases to re-create the entire collection (bulk delete)
  - –However, it's the place when hibernate could do better, having all required data available:
    - Original size, added and removed elements
    - Min(#added + #removed, 1 + #size - #removed + #added)

Puzzle #3

Fashionable Developer

Fashionable Developer

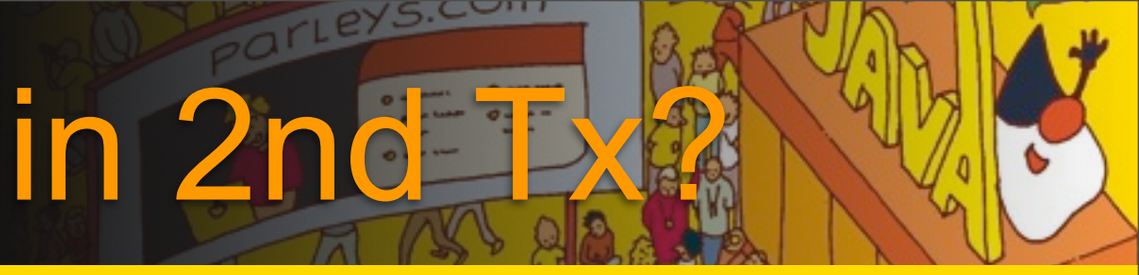
```
@Entity
public class Developer {
    @Id @GeneratedValue
    private Long id;
    private String mainTechnology;

    public boolean likesMainTechnology() {
        return "hibernate".equalsIgnoreCase(mainTechnology);
    }
}
// creates and persists a developer that uses hibernate as mainTechnology

// new EntityManager and new transaction
Developer dev = em.find(Developer.class, id);
boolean foundCoolStuff = false;
for (String tech : new String[]{"HTML5", "Android", "Scala"}) {
    dev.setMainTechnology(tech);
    // othersAreUsingIt:
    // select count(*) from Developer where mainTechnology = ? and id != ?
    if (othersAreUsingIt(tech, dev) && dev.likesMainTechnology()) {
        foundCoolStuff = true; break;
    }
}
if (!foundCoolStuff) {
    // still use hibernate
    dev.setMainTechnology("hibernate");
}
```

How Many Queries in 2nd Tx?

```
@Entity
public class Developer {
    @Id @GeneratedValue
    private Long id;
    private String mainTechnology;

    public boolean likesMainTechnology() {
        return "hibernate".equalsIgnoreCase(mainTechnology);
    }
}
// creates and persists a developer that uses hibernate as mainTechnology

// new EntityManager and new transaction
Developer dev = em.find(Developer.class, id);
boolean foundCoolStuff = false;
for (String tech : new String[]{"HTML5", "Android", "Scala"}) {
    dev.setMainTechnology(tech);
    // othersAreUsingIt:
    // select count(*) from Developer where mainTechnology = ? and id != ?
    if (othersAreUsingIt(tech, dev) && dev.likesMainTechnology()) {
        foundCoolStuff = true; break;
    }
}
if (!foundCoolStuff) {
    // still use hibernate
    dev.setMainTechnology("hibernate");
}
```

(a) 2 select
(b) 4 selects
(c) 4 selects, 1 update
(d) 4 selects, 4 updates
(e) None of the above

How Many Queries in 2nd Tx?



(a) 2 selects

(b) 4 selects

(c) 4 selects, 1 update

(d) 4 selects, 4 inserts

(e) None of the above

Hibernate must guarantee correctness of executed queries, therefore in certain situations it must perform flushes during a transaction.

Fashionable Developer

```
@Entity
public class Developer {
    @Id @GeneratedValue
    private Long id;
    private String mainTechnology;

    public boolean likesMainTechnology() {
        return "hibernate".equalsIgnoreCase(mainTechnology);
    }
}
// creates and persists a developer that uses hibernate as mainTechnology

// new EntityManager and new transaction
Developer dev = em.find(Developer.class, id);
boolean foundCoolStuff = false;
for (String tech : new String[]{"HTML5", "Android", "Scala"}) {
    dev.setMainTechnology(tech);
    // othersAreUsingIt:
    // select count(*) from Developer where mainTechnology = ? and id != ?
    if (/*othersAreUsingIt(tech, dev) &&*/ dev.likesMainTechnology()) {
        foundCoolStuff = true; break;
    }
}
if (!foundCoolStuff) {
    // still use hibernate
    dev.setMainTechnology("hibernate");
}
```

NO UPDATES

Anti-Pattern #3

Temporary Changes

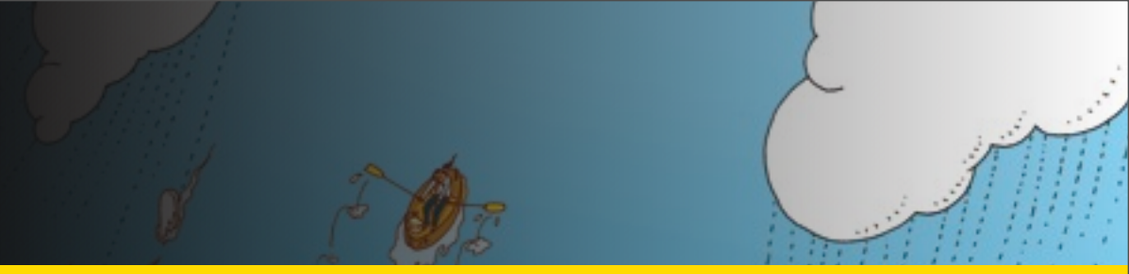
Temporary Changes

- Description
  - We change the persistent data of an object temporarily to test some features
  - In the meantime a database flush occurs triggered, for example, by a HQL touching the table
- Consequences
  - Flushes can be performed during the transaction, not only at the commit time
  - Temporary changes persisted into the database unnecessarily
- Solution
  - Plan your business model and processing to avoid temporary changes to your persistent data

Puzzle #4

Plant a Tree

Plant a Tree



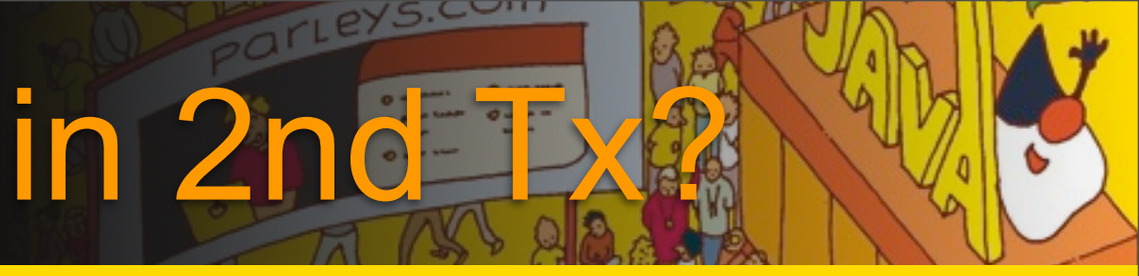
```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    Collection<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

How Many Queries in 2nd Tx?

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    Collection<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

(a) 1 select, 2 inserts
(b) 2 selects, 2 inserts
(c) 2 selects, 1 delete, 10.000+2 inserts
(d) Even more ;-)

How Many Queries in 2nd Tx?



(a) 1 select, 2 inserts

(b) 2 selects, 2 inserts

(c) 2 selects, 1 delete, 10.000+2 inserts

(d) Even more ;-)

The combination of **OneToMany** and **Collection** enables a bag semantic. That's why the collection is re-created.

Plant a Tree - Revisited

```
@Entity
public class Orchard {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    List<Tree> trees = new ArrayList<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“apple tree”);
em.persist(tree);
Orchard orchard = em.find(Orchard.class, id);
orchard.plantTree(tree);
```

Plant a Tree - Revisited

```
@Entity
public class Orchard {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    List<Tree> trees = new ArrayList<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“apple tree”);
em.persist(tree);
Orchard orchard = em.find(Orchard.class, id);
orchard.plantTree(tree);
```

STILL BAG SEMANTIC!

Plant a Tree - Revisited

```
@Entity
public class Orchard {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    List<Tree> trees = new ArrayList<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“apple tree”);
em.persist(tree);
Orchard orchard = em.find(Orchard.class, id);
orchard.plantTree(tree);
```

STILL BAG SEMANTIC!

Use OrderColumn or IndexColumn for list semantic.

OneToMany Mapping

Semantic	Java Type	Annotation
bag semantic	java.util.Collection java.util.List	@ElementCollection || @OneToMany || @ManyToMany
set semantic	java.util.Set	@ElementCollection || @OneToMany || @ManyToMany
list semantic	java.util.List	(@ElementCollection || @OneToMany || @ManyToMany) && (@OrderColumn || @IndexColumn)

@OneToMany (no cascade option)

OneToMany Mapping

	add element	remove element	update element
bag semantic	re-create: 1 delete + N inserts	re-create: 1 delete + N inserts	1 update
set semantic	1 insert	1 delete	1 update
list semantic	1 insert + M updates	1 delete + M updates*	1 update

@OneToMany (no cascade option)

OneToMany Mapping

	add element	remove element	update element
bag semantic	re-create: 1 delete + N inserts	re-create: 1 delete + N inserts	1 update
set semantic	1 insert	1 delete	1 update
list semantic	1 insert + M updates	1 delete + M updates*	1 update

@OneToMany (no cascade option)

OneToMany Mapping

	add element	remove element	update element
set semantic	1 insert	1 delete	1 update
list semantic	1 insert + M updates	1 delete + M updates*	1 update

Oops, we have a problem.
(list: removal of nth element, n < size-2)

```
Hibernate: update Forest_Tree set trees_id=? where Forest_id=? and trees_ORDER=?
2832 [main] WARN  org.hibernate.util.JDBCExceptionReporter  - SQL Error: 1062, SQLState: 23000
2832 [main] ERROR org.hibernate.util.JDBCExceptionReporter  - Duplicate entry '10' for key 'trees_id'
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
    at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:93)
    at com.yonita.examples.jpapuzzles.JPA.execute(JPA.java:23)
    at com.yonita.examples.jpapuzzles.puzzle5.collection.Forest.main(Forest.java:68) <5 internal calls>
Caused by: javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1387)
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1315)
    at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:81)
    ... 7 more
Caused by: org.hibernate.exception.ConstraintViolationException: Could not execute JDBC batch update
```

@OneToMany (no cascade option)

Anti-Pattern #4

Inadequate Collection Type (Owning Side)

Inadequate Collection Mapping (Owning Side)

- Description
  - Chosen an inadequate collection mapping on the owning side
- Consequences
  - Additional queries executed
- Solution
  - Consider your use cases and usage patterns and choose a mapping with the least performance overhead

Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany
    Set<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree oak = new Tree(“oak”);
em.persist(oak);
Forest forest = em.find(Forest.class, id);
forest.plantTree(oak);
```

Anti-Pattern #5

OneToMany as Owning Side

OneToMany as Owning Side

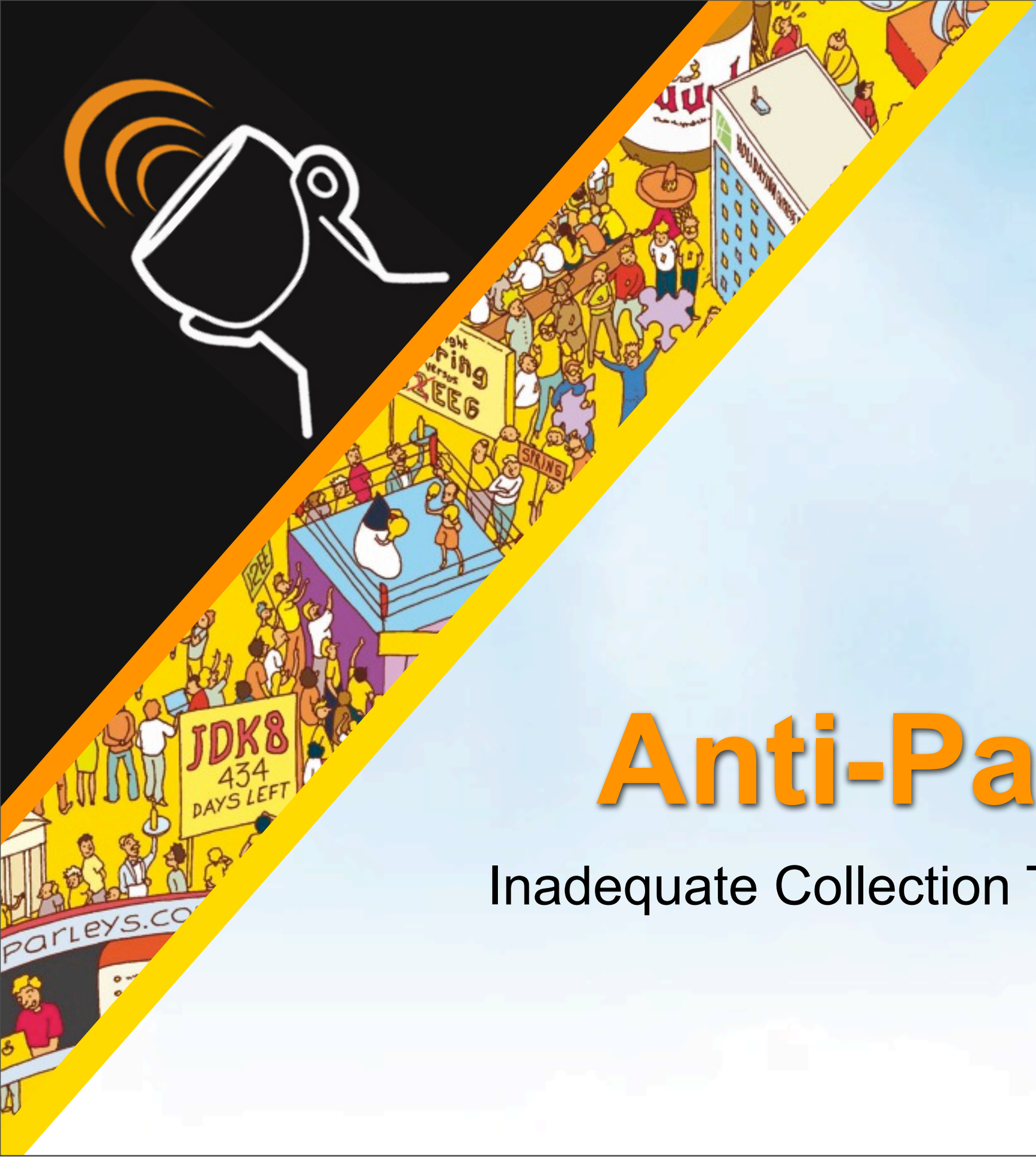
- Description
  - –OneToMany used as the owning side of a relationship
- Consequences
  - –Collection elements loaded into memory
  - –Possibly unnecessary queries
  - –Transaction and locking schema problems: version, optimistic locking
- Solution
  - –Use ManyToOne as the owning side and OneToMany as the inverse side
  - –Or at least exclude the collection from locking
    - @OptimisticLock(excluded=true)

Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Set<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    public void setForest(Forest forest) {
        this.forest = forest;
        this.forest.plantTree(this);
    }
}
```

Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Set<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    public void setForest(Forest forest) {
        this.forest = forest;
        this.forest.plantTree(this);
    }
}
```



Anti-Pattern #6

Inadequate Collection Type (Inverse Side)

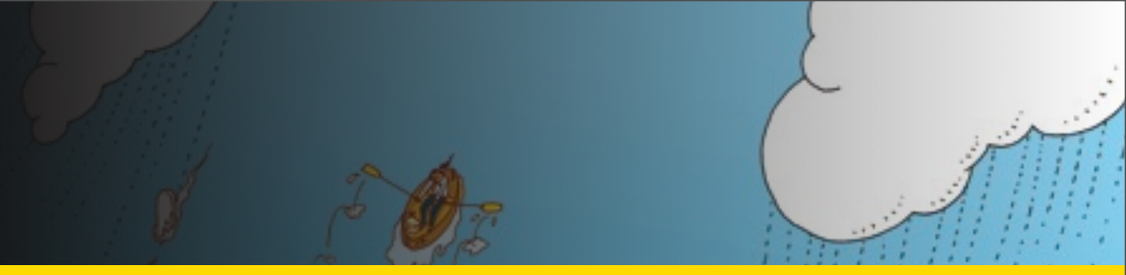
Inadequate Collection Type (Inverse Side)

- Description
  - Set used as a collection type on the inverse side of a relationship and, to keep the model consistent, the owning side updates the inverse side
- Consequences
  - Due to a set semantic, all elements must be loaded when we add a new element
- Solution
  - Use Collection or List instead

Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Collection<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    public void setForest(Forest forest) {
        this.forest = forest;
        this.forest.plantTree(this);
    }
}
```

Plant a Tree



```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Collection<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}

@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    // ...
}
// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
em.remove(forest);
```

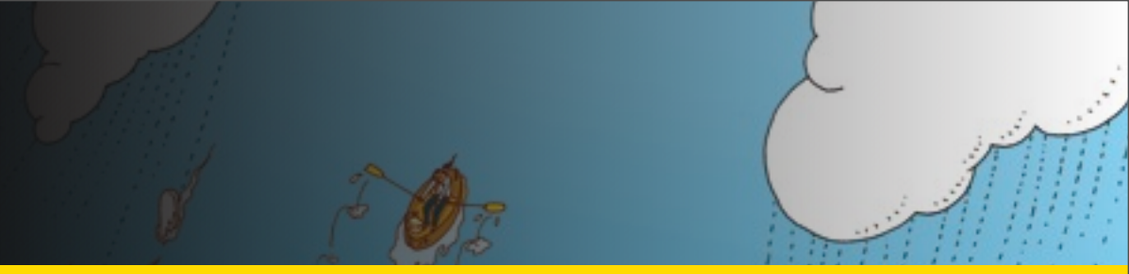
Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Collection<Tree> trees = new HashSet<Tree>();

    public void plantTree(Tree tree) {
        trees.add(tree);
    }
}
```

```
    ... 7 more
Caused by: java.sql.BatchUpdateException: Cannot delete or update a parent row: a foreign key constraint fails (`jpapuzzles`.`tree`
    at com.mysql.jdbc.PreparedStatement.executeBatchSerially(PreparedStatement.java:2020)
    at com.mysql.jdbc.PreparedStatement.executeBatch(PreparedStatement.java:1451)
    at org.hibernate.jdbc.BatchingBatcher.doExecuteBatch(BatchingBatcher.java:70)
    at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatcher.java:268)
```

```
@E
pu
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    // ...
}
// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
em.remove(forest);
```

Plant a Tree



```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Collection<Tree> trees = new HashSet<Tree>();

    // ...
}

@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    // ...
}
// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
for (Tree tree : forest.getTrees()) {
    tree.setForest(null);
}
em.remove(forest);
```

Plant a Tree

```
@Entity
public class Forest {
    @Id @GeneratedValue
    private Long id;
    @OneToMany (mappedBy = “forest”)
    Collection<Tree> trees = new HashSet<Tree>();

    // ...
}

@Entity
public class Tree {
    @Id @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne
    Forest forest;

    // ...
}
// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
for (Tree tree : forest.getTrees()) {
    tree.setForest(null);
}
em.remove(forest);
```

10.000 UPDATES
1 DELETE

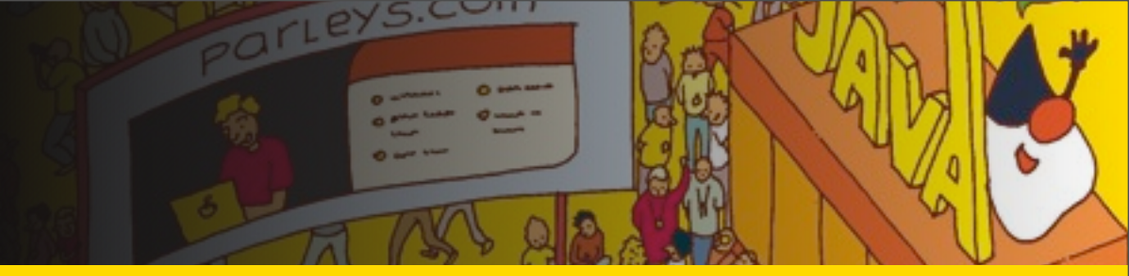
Anti-Pattern #7

One by One Processing

Inadequate Collection Type (Inverse Side)

- Description
  - Sequential processing of a persistent collection/objects
- Consequences
  - Each database operation performed separately for each element
- Solution
  - Bulk queries
  - That's why CaveatEmptor was implemented like this (aggregate function on a database level)

Conclusions



- Big data cause big problems
  - –Standard mappings don't handle large datasets well
  - –Smart model, bulk processing, projections
- RTFM
  - –Most of information can be found in the hibernate docs
  - –Understand what you read
  - –Think about consequences
  - –And hopefully you'll be fine :)
- Hibernate
  - –Smarter policies and data structures

Contact

- email: patrycja@yonita.com
- twitter: @yonlabs
- www: http://www.yonita.com