Java Persistence with Hibernate

Outline

- What is Hibernate?
- Why Hibernate?
- What are the alternatives to Hibernate?
- Object relational mapping issues
- Simple Hibernate example
- Hibernate Power
  - Advanced association mapping
  - Advanced inheritance mapping
  - Flexibility and extensibility
- Hibernate design consideration (tips and tricks)

What is hibernate?

- Hibernate is an object-relational mapping framework (established in 2001)
- In English, that means it provides tools to get java objects in and out of DB
    - It also helps to get their relationships to other objects in and out of DB
- It does so without a lot of intrusion
    - You don’t have to know a lot of hibernate API
    - You don’t have to know or use a lot of SQL

Why use hibernate?

- Too much developer time is spent getting data in and out of the database and objects (Java Objects)
- Take a guess !

That’s
30% – 70%
Of your developer’s time

Why Hibernate …

- Hibernate's stated goal "is to relieve the developer from 95 percent of common data persistence related programming tasks"
  - Conservative numbers – 95% of 30% >= 25% increase in productivity
- Hibernate can also help your application be more platform independent/portable
  - Database access configuration and SQL are moved to configuration files
- Hibernate can help with performance
  - It generates efficient queries consistently

Alternatives

- Hibernate is certainly not the only ORM framework
- There are lot more
  - Java Persistence API – From Sun (mature now, but was only a specification when hibernate is used in industry)
  - Java Data Objects – Requires additional compilation steps that alters .class files
  - EJB Container Managed Persistence (CMP) – Pretty complex technology but was the sole provider of ORB before the evolution of other ORM framework and got lot of container specific features, cannot be used in standalone
  - iBatis – Requires lot of SQL skills
  - TopLink – Is a commercial product now owned by Oracle
  - Castor –Open Source ORM but not as popular as hibernate

Why so many ORMs?

- Mapping objects to relational database is hard
- The two systems satisfy different need
- There are a number of obstacles to overcome based on the differences
  - Identity
  - Granularity
  - Associations
  - Navigation
  - Inheritance
  - Data type mismatches
- All above needs to be overcome for an ORM to be successful

Identity

- A row is uniquely identified from all other rows by its identity
- An object’s identity doesn’t always translate well to the concept of primary key
- In Java, what is an object’s identity? Its data or its place in memory?
  - Consider the following: are two objects equal or identical?

```
accountA.equals(accountB)
accountA == accountB
```

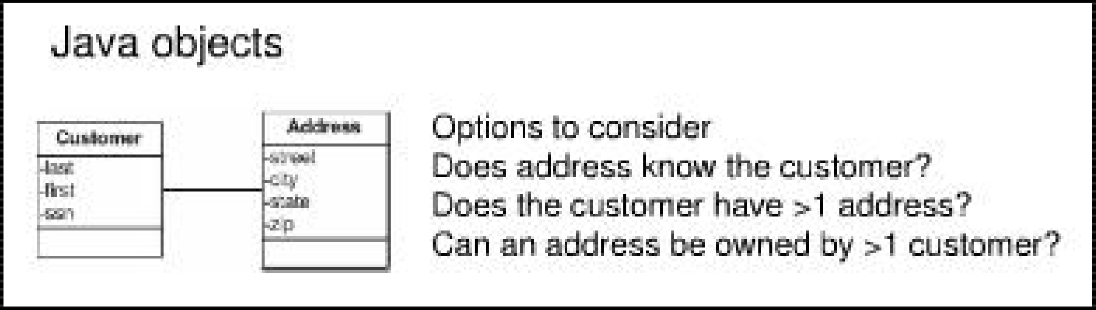
Granularity

- Objects and tables are modeled with different levels of granularity
- Table structures are often “de-normalized” in order to support better performance
  - Table rows end up mapping to multiple objects
- Objects tend to be more fine-grained

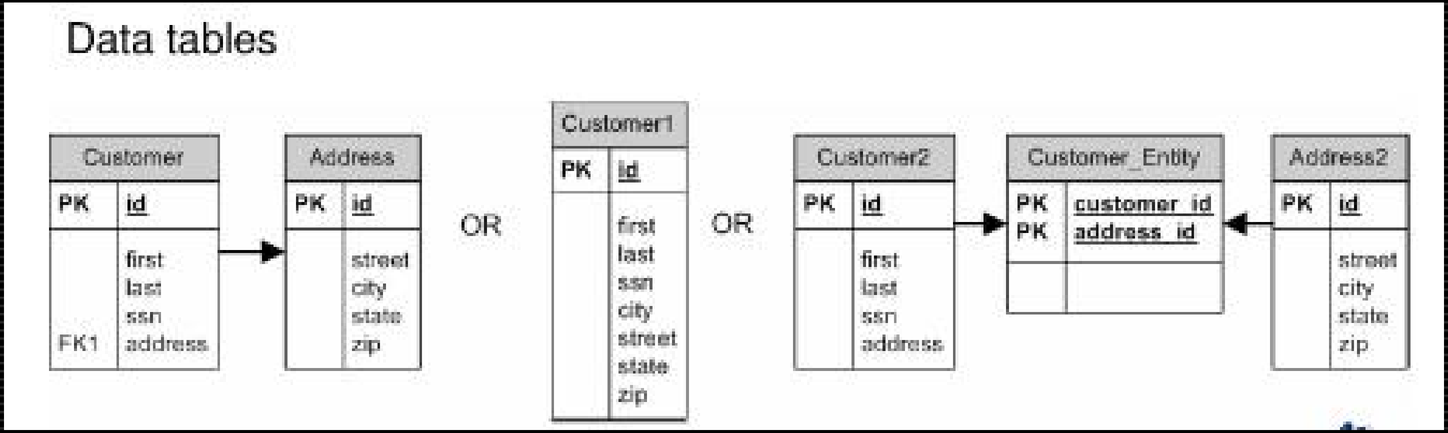
Associations

- Associations in Java are either unidirectional or bidirectional
  - Based on attributes and accessors
  - Cardinality is represented by complex types (List, Set)
  - Object identification can impact certain associations (Set)
- Relating data in database tables is based on joins
  - There is no concept of directionality
  - Many to Many relation require the addition of a join table
  - Identity, especially foreign key identity, greatly impacts associations.

Association example

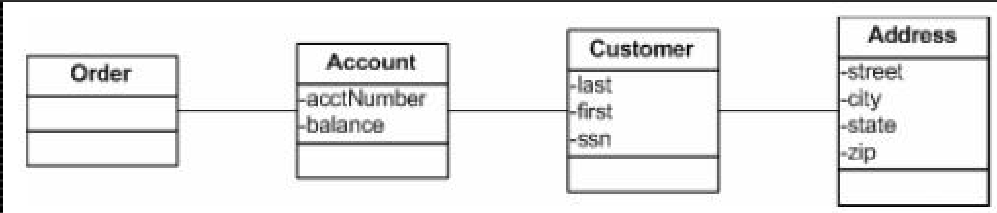


Options to consider
Does address know the customer?
Does the customer have >1 address?
Can an address be owned by >1 customer?



Navigation and Association Traversal

- Navigating Java Objects to get property information often required traversal of the object graph
  - This can be quite expensive if the objects are distributed or need to be created before traversed
  - Consider the below



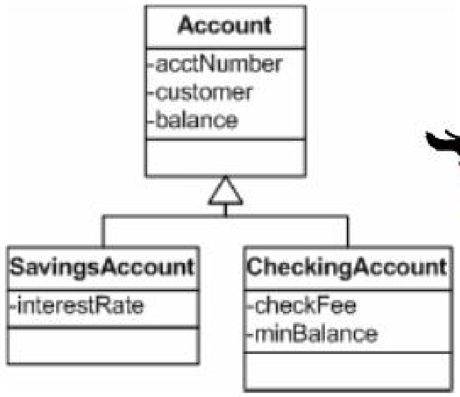
```
anAcct = anOrder.getAccount();
aCust = anAcct.getCustomer();
orderAddress = aCust.getAddress();
shipTo(orderAddress);
```

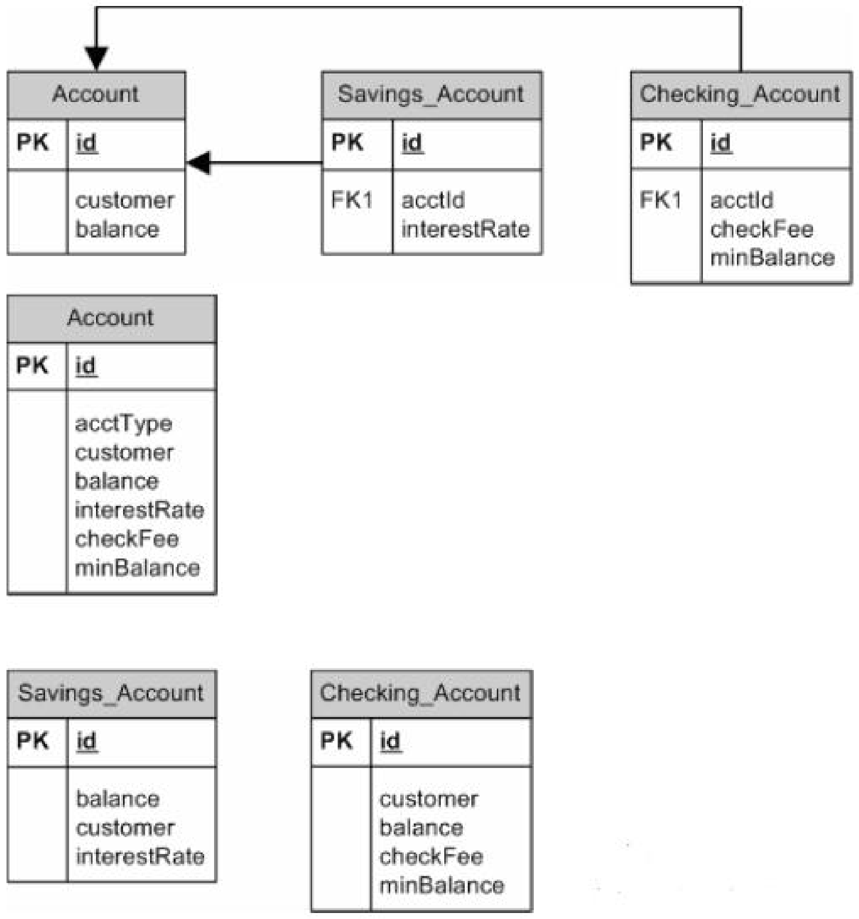
- Navigating in database is handled, potentially, by a single join
  - select a.street, a.city, a.state, a.zip from address a, order o, account t, customer c where o.id=t.order_id and t.customer_id=c.id and c.address_id=a.id

Inheritance

- Inheritance and Polymorphism are important concepts in OO programming languages like Java
  - Add power and flexibility to our programming environments
- Databases have no equivalent concepts to inheritance
- Mapping inheritance trees in Java to the database creates an opportunity for creativeness

Inheritance example

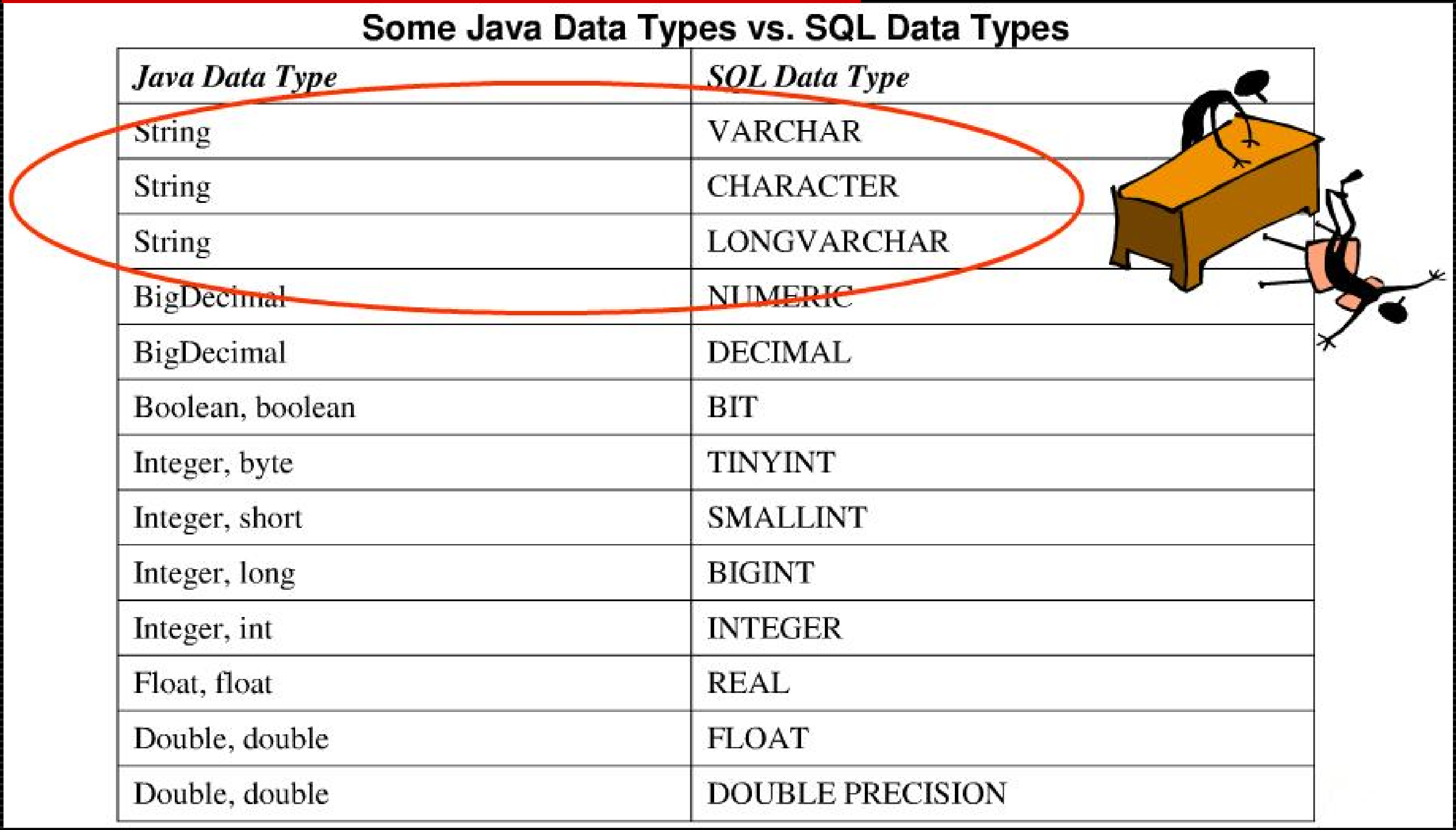




Datatypes

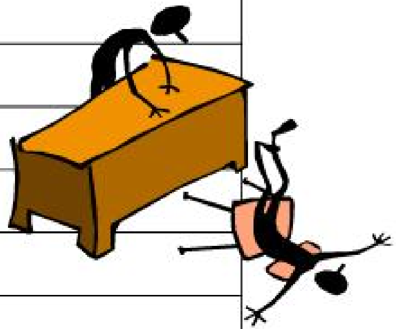
- Even simple types are not easily mapped between Java and the major relational databases
  - Do you remember some case?

Datatypes Mapping example



Some Java Data Types vs. SQL Data Types

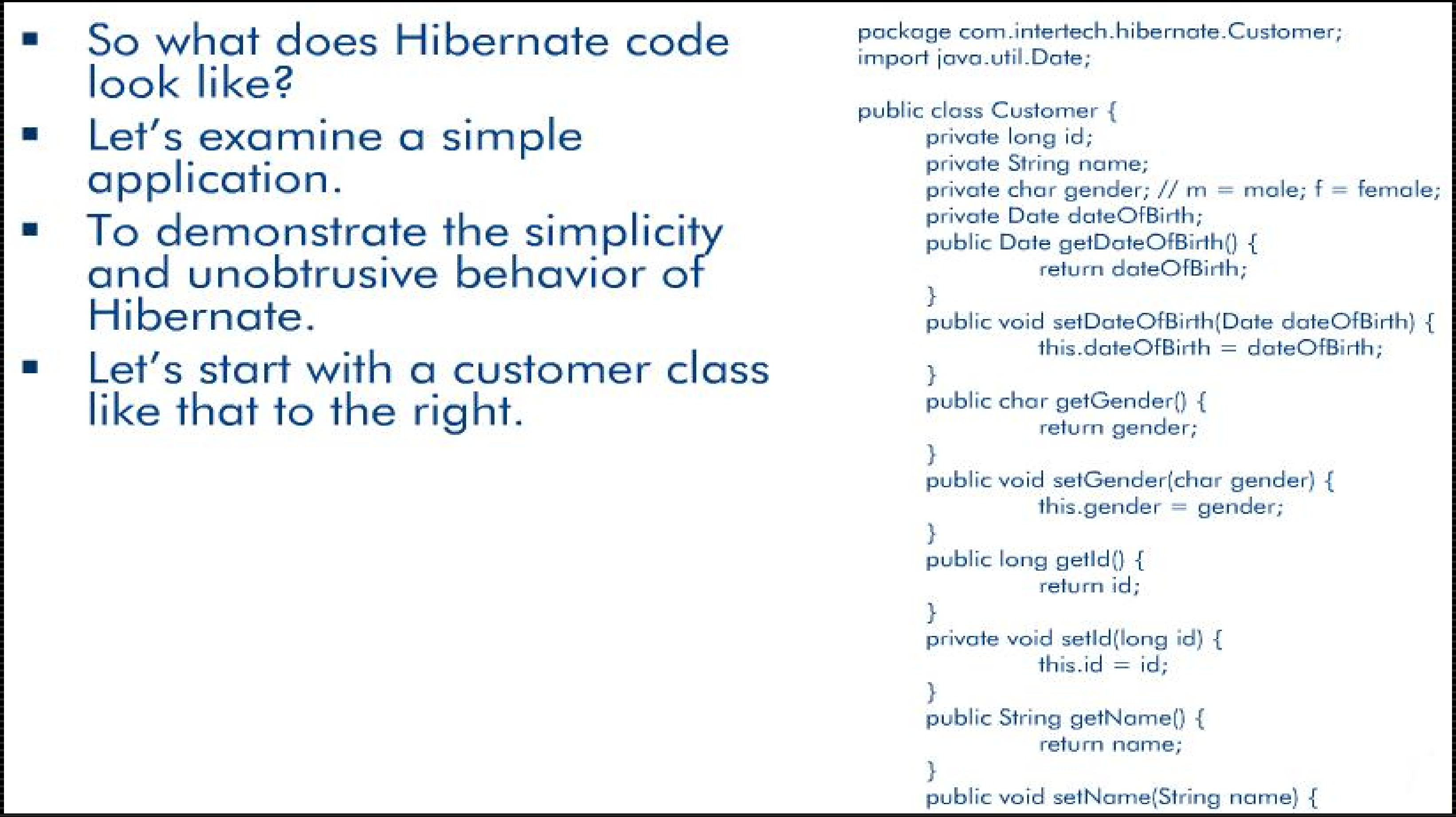
Java Data Type	*SQL Data Type*
String	VARCHAR
String	CHARACTER
String	LONGVARCHAR
BigDecimal	NUMERIC
BigDecimal	DECIMAL
Boolean, boolean	BIT
Integer, byte	TINYINT
Integer, short	SMALLINT
Integer, long	BIGINT
Integer, int	INTEGER
Float, float	REAL
Double, double	FLOAT
Double, double	DOUBLE PRECISION



Hibernate to the rescue

- Hibernate handles all of these issues
- In fact, Hibernate provides several options in most cases to allow you to handle them in a variety of ways
- Why the options?
  - The database may have come before the application (lay on top of legacy systems)
  - The application may drive the development of the database (green field applications)
  - The database and application must be integrated (meet-in-the-middle approach)

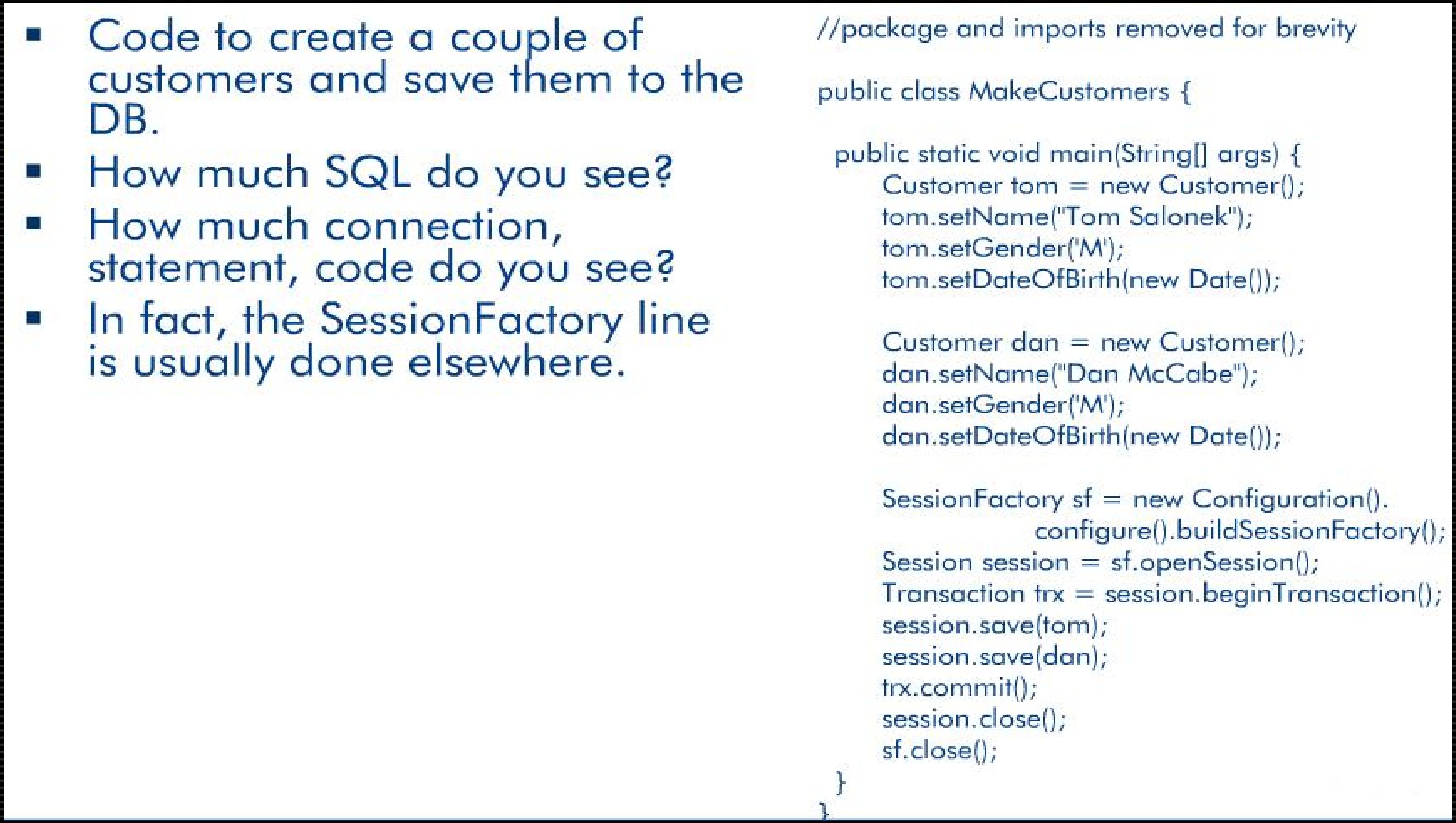
A Simple Example of Hibernate



- So what does Hibernate code look like?
- Let's examine a simple application.
- To demonstrate the simplicity and unobtrusive behavior of Hibernate.
- Let's start with a customer class like that to the right.

```
package com.intertech.hibernate.Customer;
import java.util.Date;

public class Customer {
        private long id;
        private String name;
        private char gender; // m = male; f = female;
        private Date dateOfBirth;
        public Date getDateOfBirth() {
                return dateOfBirth;
        }
        public void setDateOfBirth(Date dateOfBirth) {
                this.dateOfBirth = dateOfBirth;
        }
        public char getGender() {
                return gender;
        }
        public void setGender(char gender) {
                this.gender = gender;
        }
        public long getId() {
                return id;
        }
        private void setId(long id) {
                this.id = id;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
```

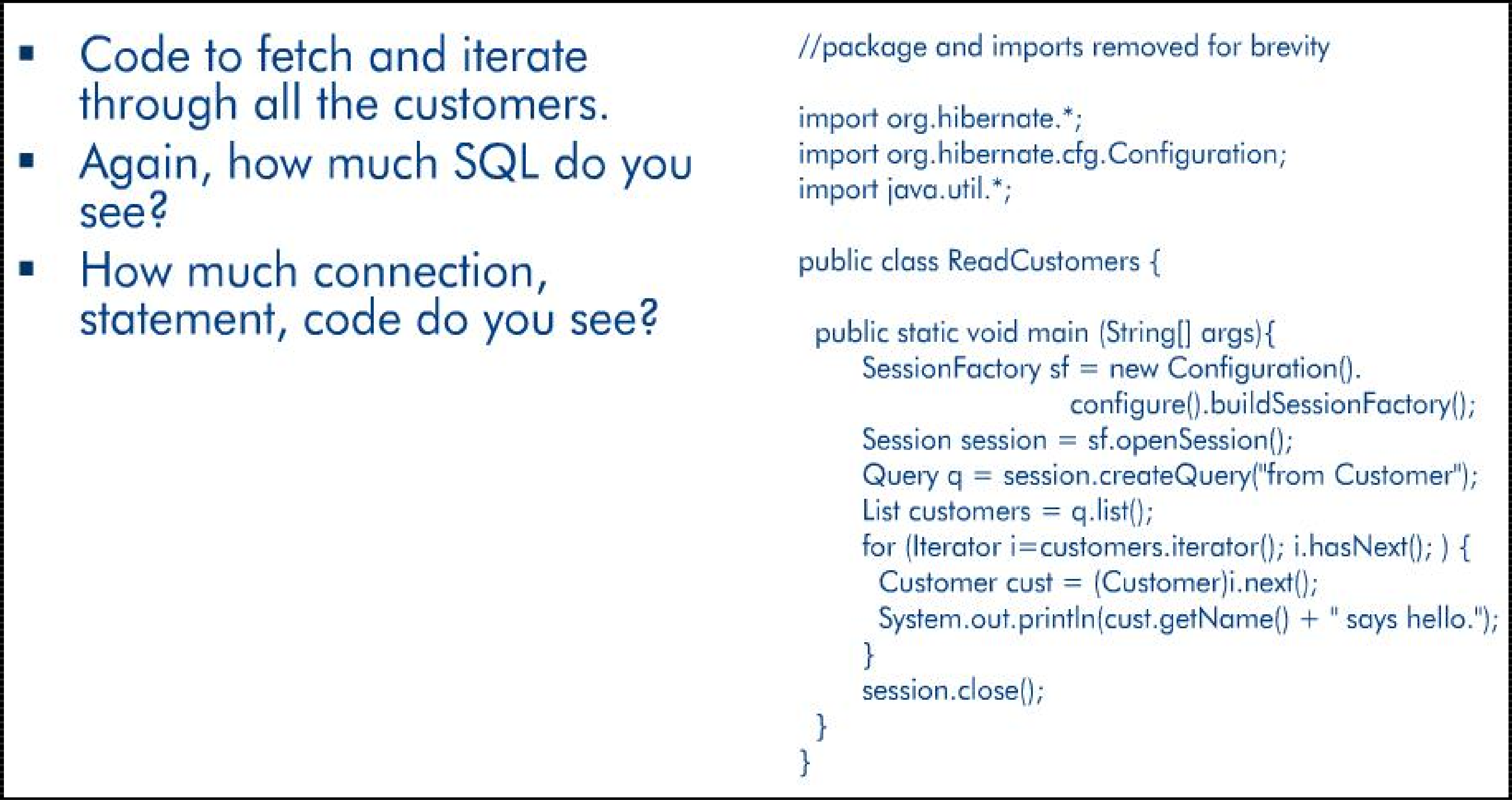
Creating and Saving



- Code to create a couple of customers and save them to the DB.
- How much SQL do you see?
- How much connection, statement, code do you see?
- In fact, the SessionFactory line is usually done elsewhere.

```
//package and imports removed for brevity

public class MakeCustomers {

  public static void main(String[] args) {
      Customer tom = new Customer();
      tom.setName("Tom Salonek");
      tom.setGender('M');
      tom.setDateOfBirth(new Date());

      Customer dan = new Customer();
      dan.setName("Dan McCabe");
      dan.setGender('M');
      dan.setDateOfBirth(new Date());

      SessionFactory sf = new Configuration().
                  configure().buildSessionFactory();
      Session session = sf.openSession();
      Transaction trx = session.beginTransaction();
      session.save(tom);
      session.save(dan);
      trx.commit();
      session.close();
      sf.close();
  }
}
```

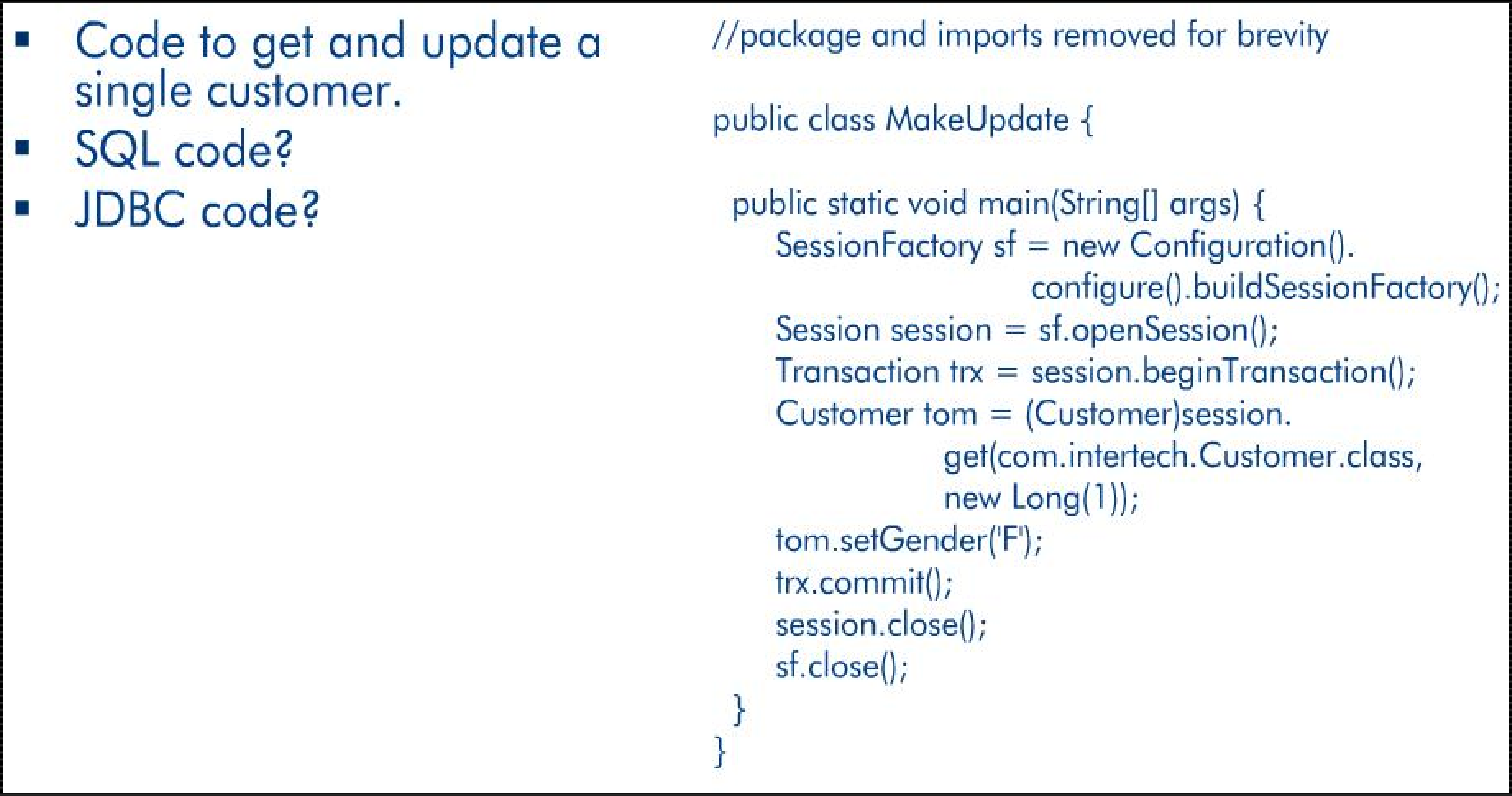
Retrieving



- Code to fetch and iterate through all the customers.
- Again, how much SQL do you see?
- How much connection, statement, code do you see?

```
//package and imports removed for brevity

import org.hibernate.*;
import org.hibernate.cfg.Configuration;
import java.util.*;

public class ReadCustomers {

 public static void main (String[] args){
     SessionFactory sf = new Configuration().
                         configure().buildSessionFactory();
     Session session = sf.openSession();
     Query q = session.createQuery("from Customer");
     List customers = q.list();
     for (Iterator i=customers.iterator(); i.hasNext(); ) {
       Customer cust = (Customer)i.next();
       System.out.println(cust.getName() + " says hello.");
     }
     session.close();
 }
}
```

Updating



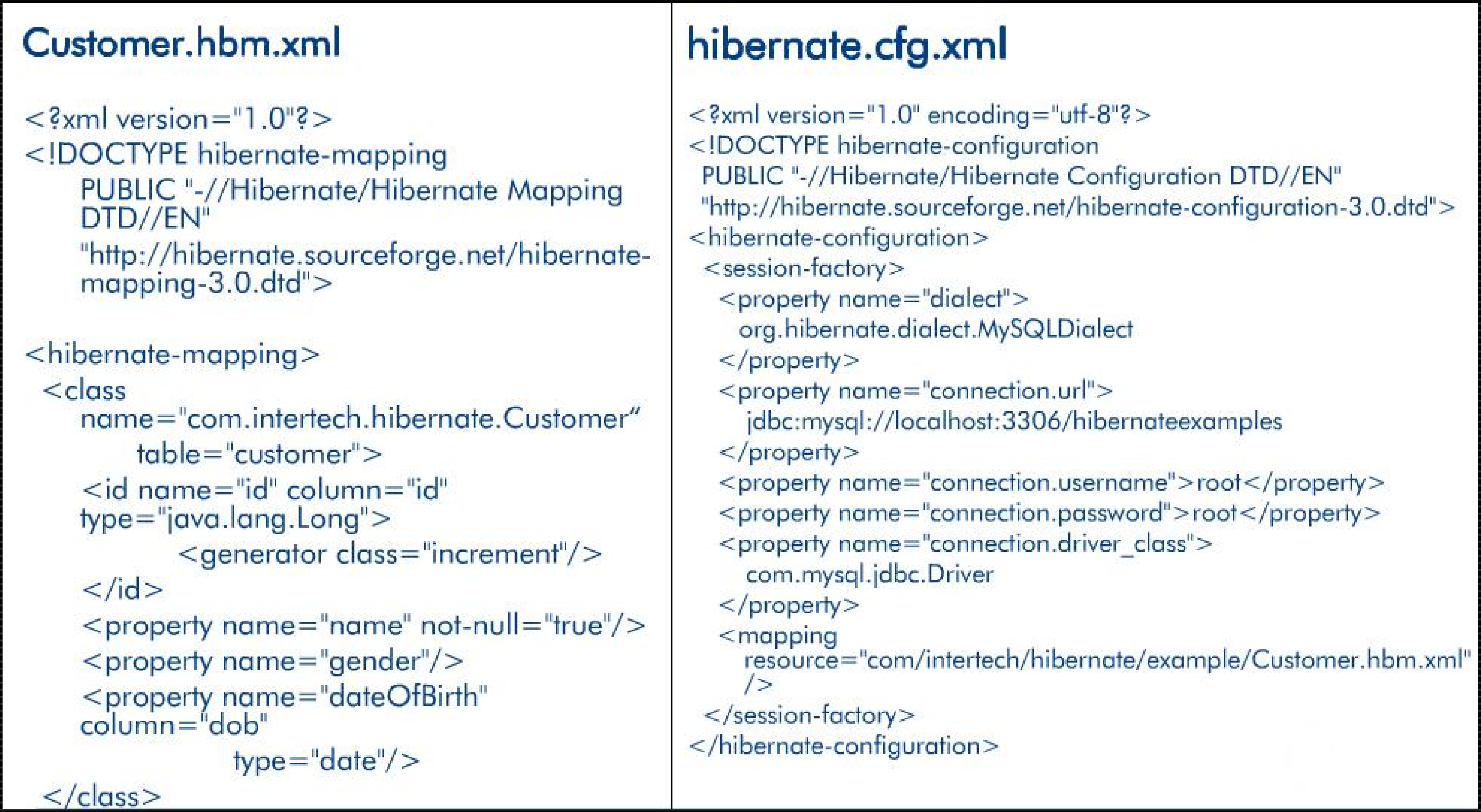
- Code to get and update a single customer.
- SQL code?
- JDBC code?

```
//package and imports removed for brevity

public class MakeUpdate {

  public static void main(String[] args) {
      SessionFactory sf = new Configuration().
                          configure().buildSessionFactory();
      Session session = sf.openSession();
      Transaction trx = session.beginTransaction();
      Customer tom = (Customer)session.
                    get(com.intertech.Customer.class,
                    new Long(1));
      tom.setGender('F');
      trx.commit();
      session.close();
      sf.close();
  }
}
```

Configuring Hibernate

- In order to get the previous examples work, two types of configuration files must also be provided
- A hibernate cfg.xml file defines all the database connection information
  - This can be done in standard property file or xml format
  - The configuration supports both managed (app server with data source) or non-managed environments
  - Allowing of easy to setup and use dev, test, prod environments
- A classname.hbm.xml maps class properties to the database table/columns
  - Traditionally one configuration per class
  - Stored with the .class files

Configuration files



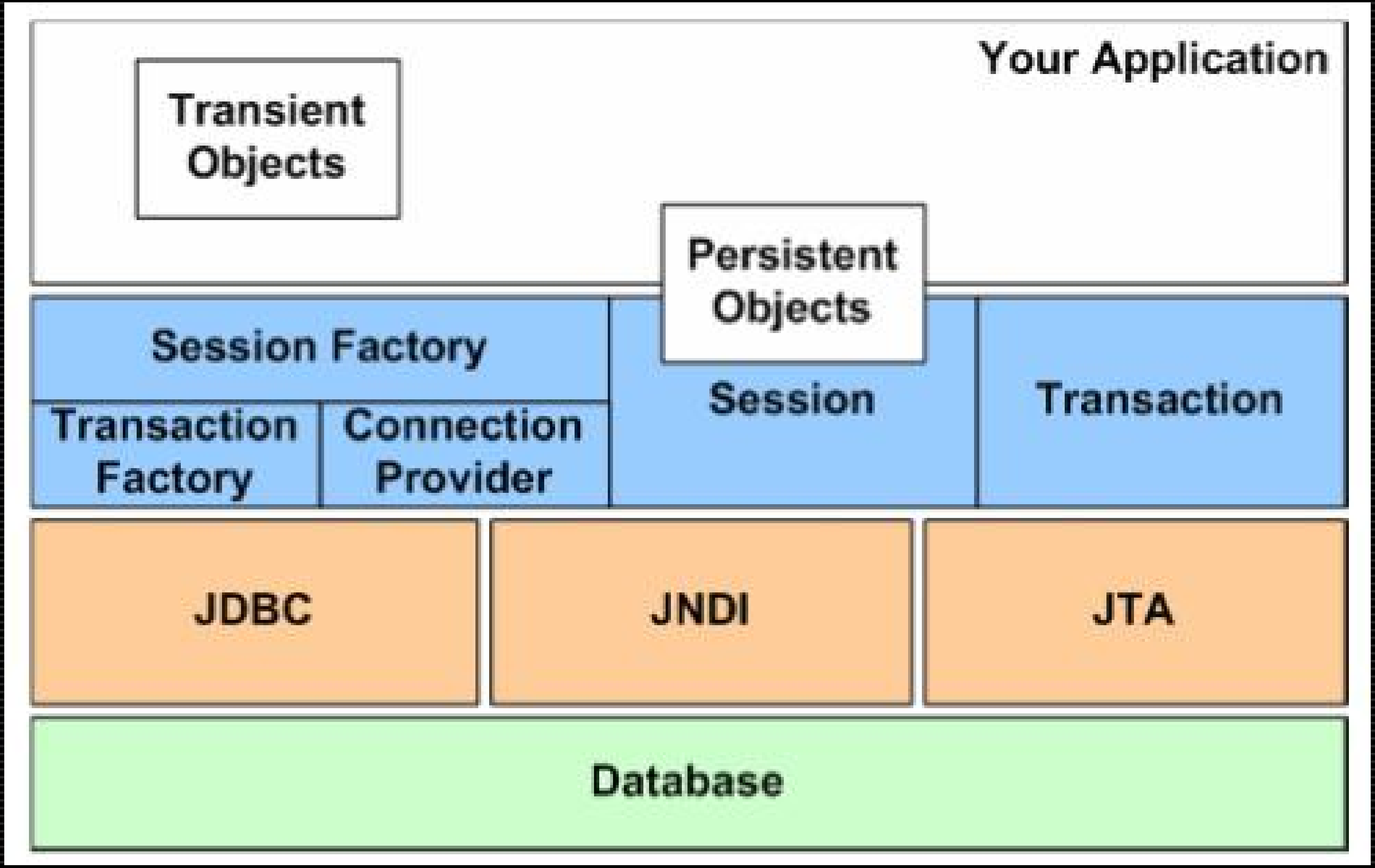
Customer.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
     PUBLIC "-//Hibernate/Hibernate Mapping
     DTD//EN"
     "http://hibernate.sourceforge.net/hibernate-
     mapping-3.0.dtd">

<hibernate-mapping>
  <class
     name="com.intertech.hibernate.Customer“
          table="customer">
     <id name="id" column="id"
     type="java.lang.Long">
               <generator class="increment"/>
     </id>
     <property name="name" not-null="true"/>
     <property name="gender"/>
     <property name="dateOfBirth"
     column="dob"
                    type="date"/>
  </class>
```

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
 PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
 "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
 <session-factory>
   <property name="dialect">
     org.hibernate.dialect.MySQLDialect
   </property>
   <property name="connection.url">
     jdbc:mysql://localhost:3306/hibernateexamples
   </property>
   <property name="connection.username">root</property>
   <property name="connection.password">root</property>
   <property name="connection.driver_class">
     com.mysql.jdbc.Driver
   </property>
   <mapping
     resource="com/intertech/hibernate/example/Customer.hbm.xml"
     />
 </session-factory>
</hibernate-configuration>
```

Hibernate Architecture and API



Configuration

- As seen in the example code, a Configuration object is the first Hibernate object you use
- The Configuration object is usually created once during application initialization
- The Configuration object reads and establishes the properties Hibernate uses to get connected
- By default, the configuration loads configuration property information and mapping files from the classpath
  - The configuration object can be told explicitly where to find files
- The Configuration object is used to create a SessionFactory and then typically discarded.

SessionFactory

- The SessionFactory is created from a Configuration object
  - SessionFactory sf = cfg.buildSessionFactory();
- The SessionFactory is an expensive object to create
  - It too is usually created during application startup
  - It should be created once and kept for later use
- The SessionFactory object is used by all the threads of an application
  - It is a thread safe object
  - One SessionFactory object is created per database
- The SessionFactory is used to create Session objects

Session

- The Session object is created from the SessionFactory object
  - Session session = sf.openSession();
- A Session object is lightweight and inexpensive to create
  - Provides the main interface to accomplish work with database
  - Does the work of getting a physical connection to database
  - Not thread safe
  - Should not be kept open for a long time
  - Applications create and destroy these objects as needed. They are created to complete a single unit of work
- When modifications are made to the database, session objects are used to create a transaction object

Transaction

- Transaction objects are obtained from a session object when a modification to the database is needed
  - Transaction trx = session.beginTransaction();
- The Transaction object provides abstraction for the underlying implementation
  - Hibernate with different transaction implementation is available (JTA, JDBC, etc)
  - It is optional, allowing developers to use their own transactional infrastructure
- Transaction object should be kept open for as short of time as possible

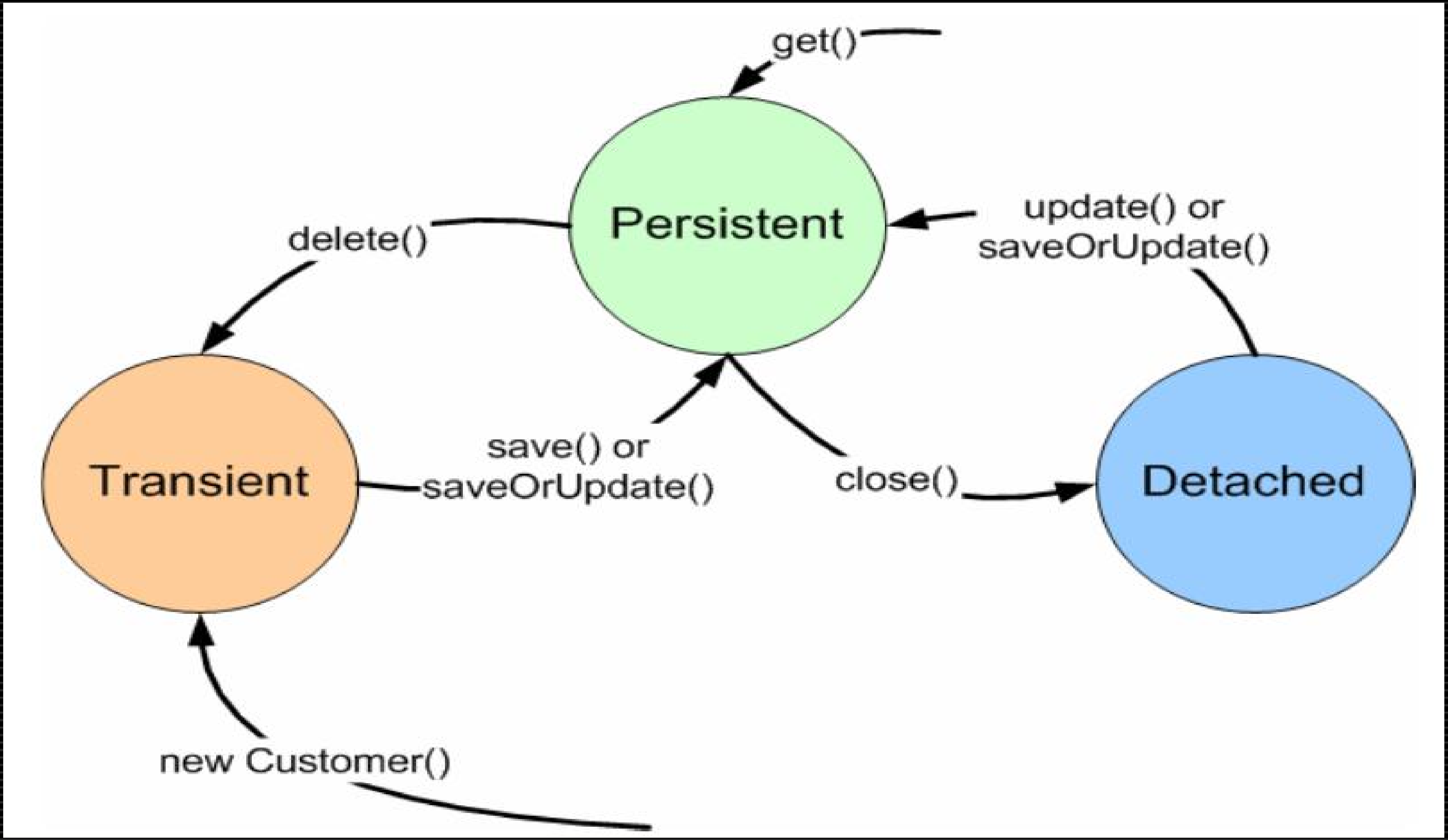
That’s the API Basics

- Configuration, SessionFactory, Session, Transaction (along with your own persistent classes) gets you started
- In fact, most of the hibernate “coding” is spent in the two configuration files
  - hibernate.cfg.xml or hibernate.properties
  - Classname.hbm.xml (one per persistent class)
- The toughest part is getting your persistent mapping set up
  - Understanding and picking the best option for properties
  - Understanding and picking the best option for associations
  - Understanding and picking the best option for your inheritance tree

Object Lifecycle

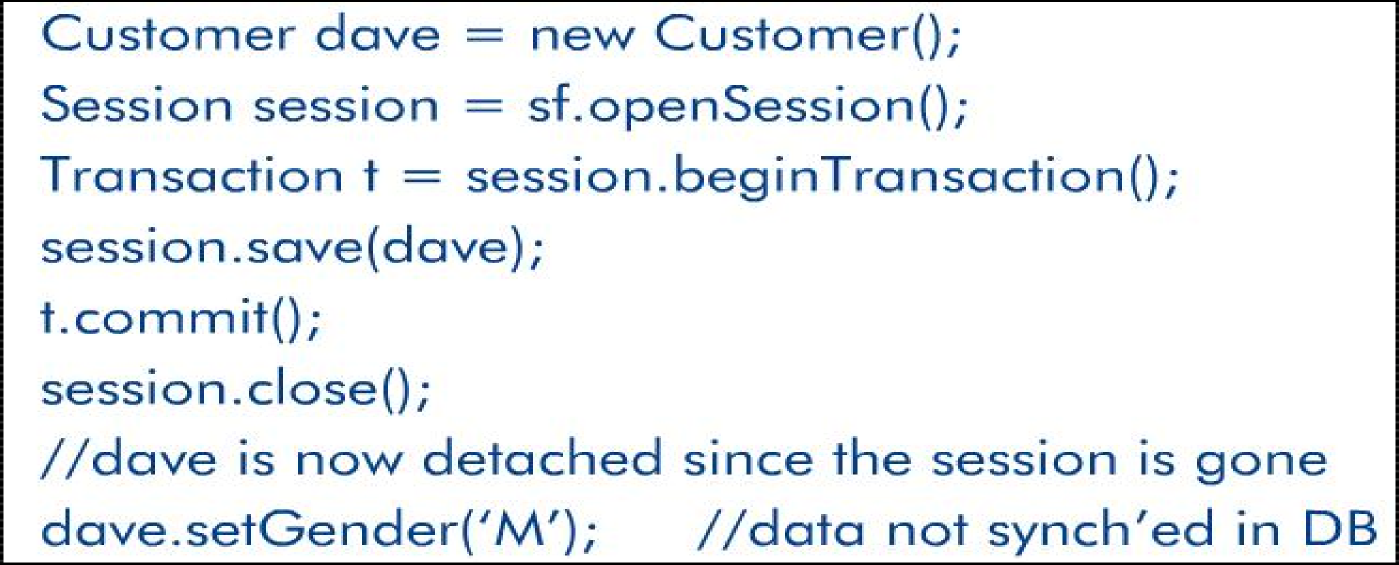
- Persistent object (like the Customer object) are synchronized with database
  - That is, the object’s state and the affected database rows are kept the same whenever either changes
- Domain objects are in one of three states
  - Transient
  - Persistent
  - Detached
- An object’s state determine if it kept synchronized with the database
- The presence of a Session and certain method calls move an object between states

Object State



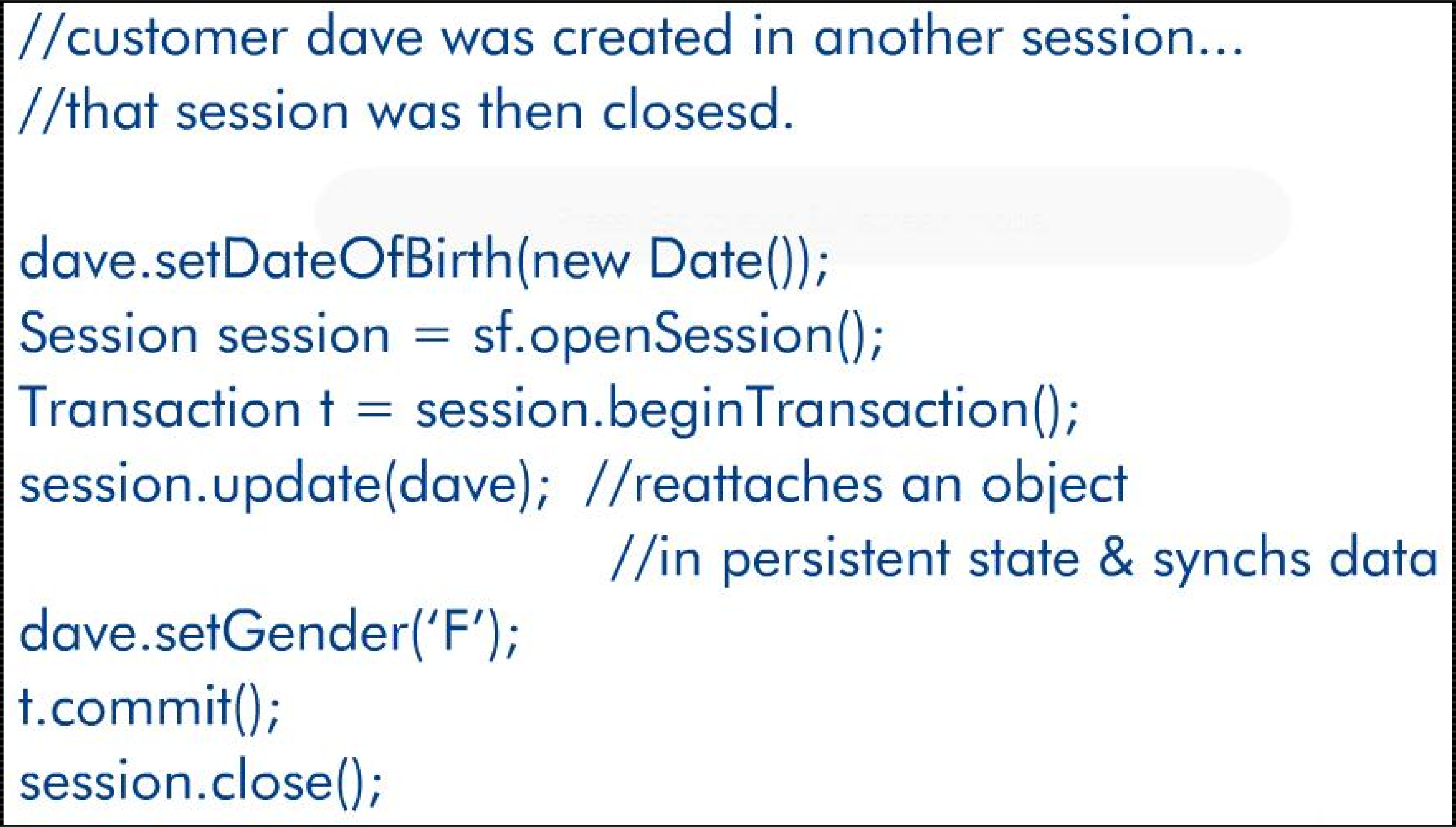
Persistent to Detached

- We have already seen an example of moving a new transient object to a persistent state
- Here is an example of an object in detached state



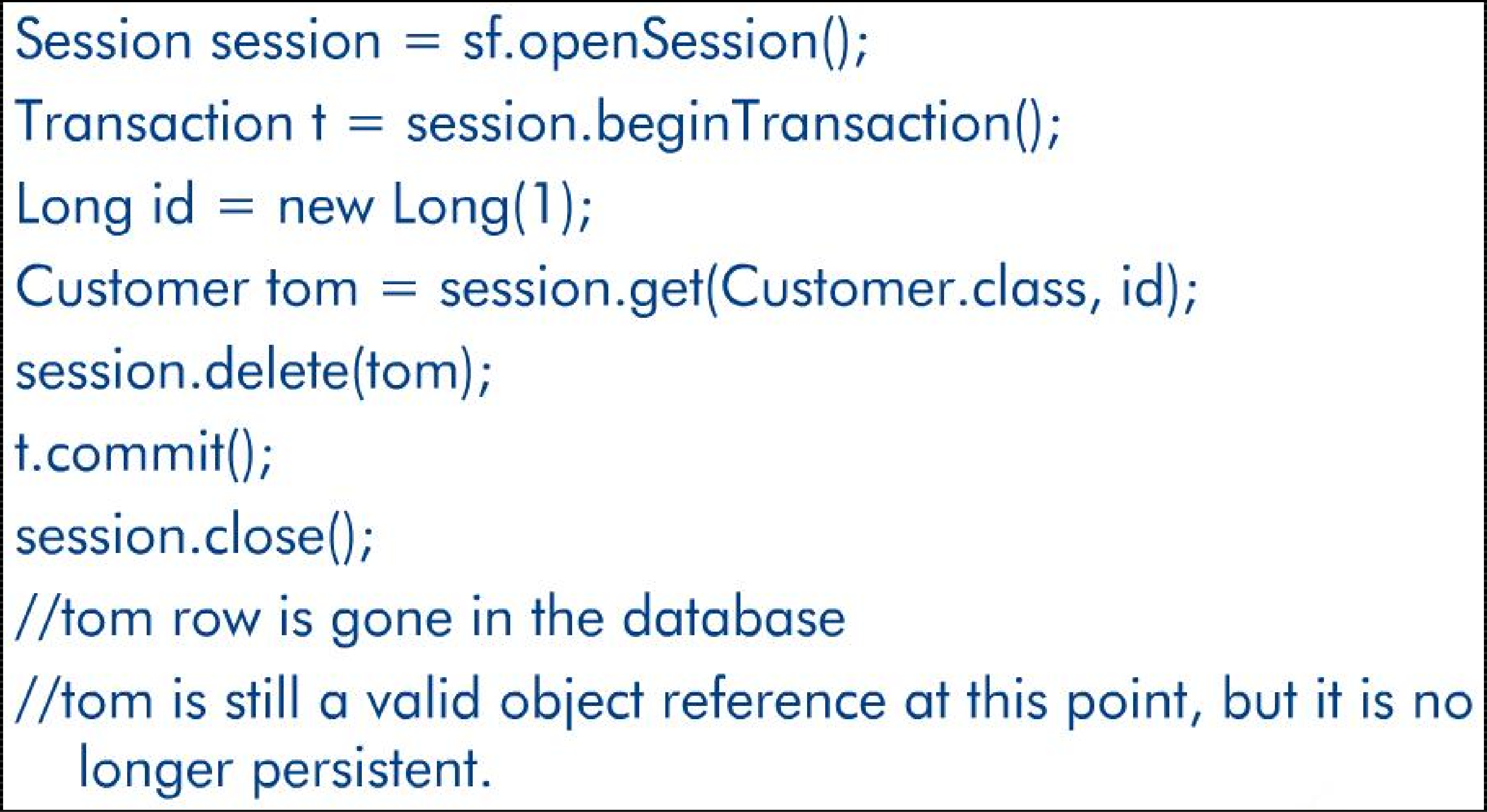
```
Customer dave = new Customer();
Session session = sf.openSession();
Transaction t = session.beginTransaction();
session.save(dave);
t.commit();
session.close();
//dave is now detached since the session is gone
dave.setGender('M');     //data not synch'ed in DB
```

Detached to Persistent



```
//customer dave was created in another session...
//that session was then closesd.

dave.setDateOfBirth(new Date());
Session session = sf.openSession();
Transaction t = session.beginTransaction();
session.update(dave);  //reattaches an object
                        //in persistent state & synchs data
dave.setGender('F');
t.commit();
session.close();
```

Back to Transient



```
Session session = sf.openSession();
Transaction t = session.beginTransaction();
Long id = new Long(1);
Customer tom = session.get(Customer.class, id);
session.delete(tom);
t.commit();
session.close();
//tom row is gone in the database
//tom is still a valid object reference at this point, but it is no
    longer persistent.
```

Associations

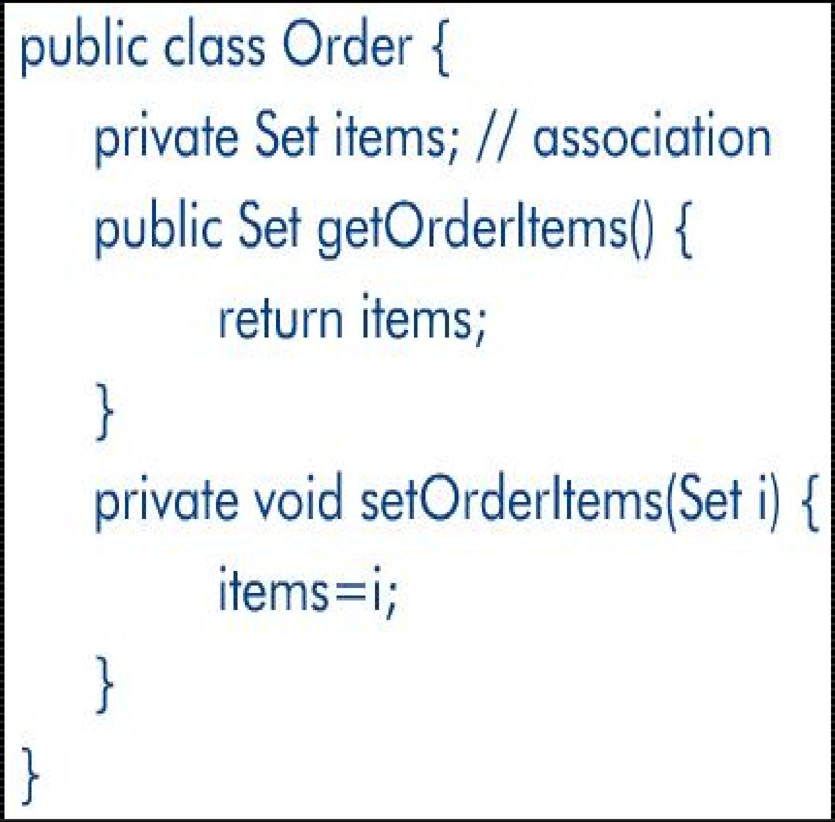
- Hibernate provides rich set of alternatives for mapping object associations in the database
- Various multiplicities of relationships
  - Many to One
  - One to Many
  - Many to Many
  - One to One
- Providing for Java’s understanding of directionality
  - Unidirectional
  - Bidirectional
- Providing for fine grained control of “transitive persistence”

Transitive Persistence

- Controls when/how associated objects react to changes in the other
  - For example, should associated address rows be deleted when a customer object is removed (cascade delete)
  - When an new order is saved should associated new line item objects also saved (cascade save)

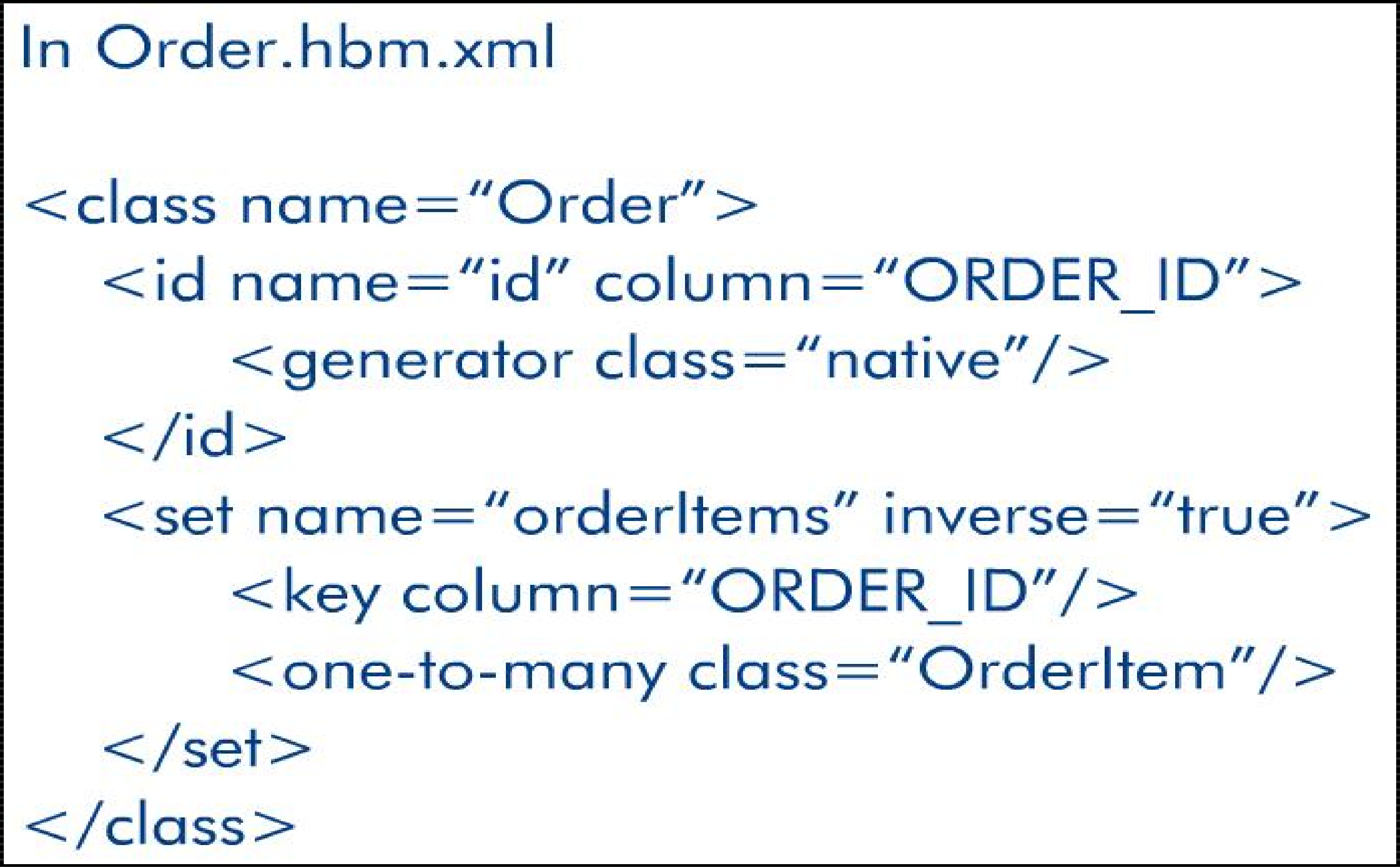
Association Mapping

- Association mapping is primarily handled in the class mapping files
- The class must provide the standard property and getter/setter methods
- For example, an Order that has an association to a set of Order Items would need definitions similar to those at right.



```
public class Order {
    private Set items; // association
    public Set getOrderItems() {
        return items;
    }
    private void setOrderItems(Set i) {
        items=i;
    }
}
```

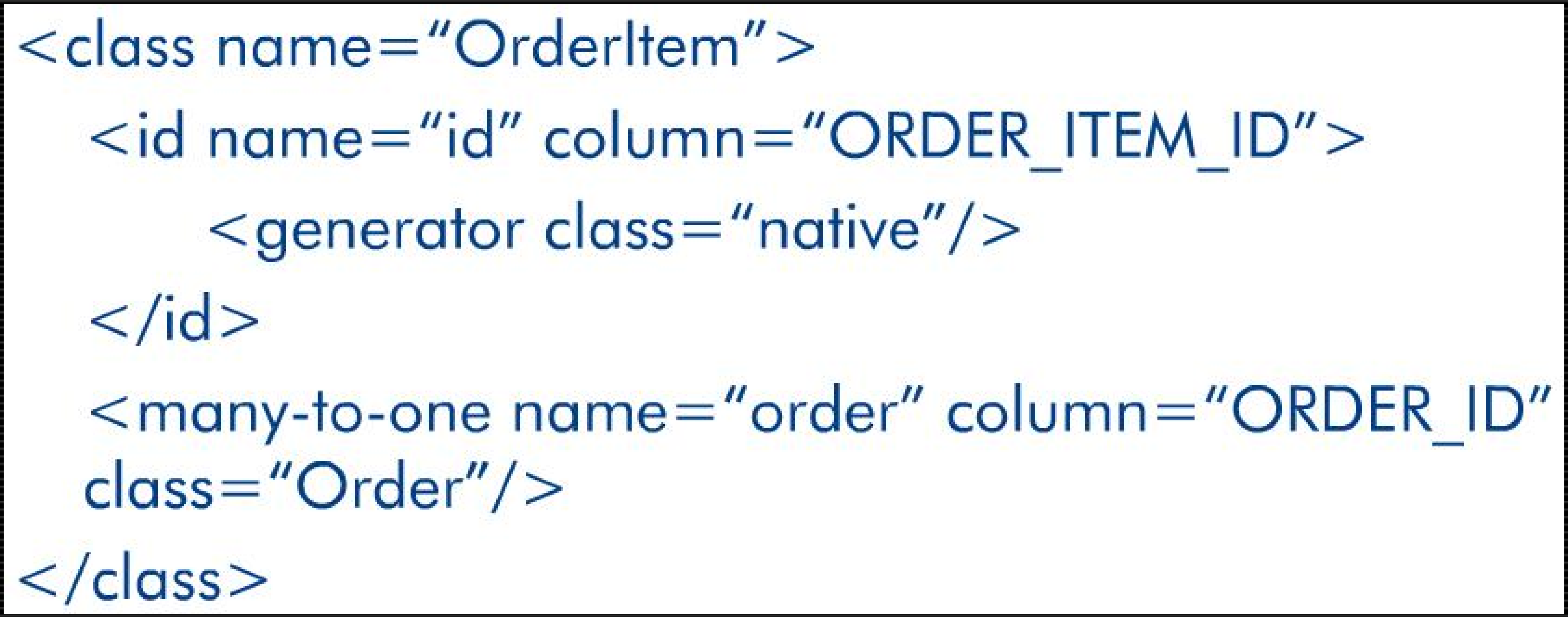
Example one-to-many mapping


In Order.hbm.xml

```
<class name="Order">
    <id name="id" column="ORDER_ID">
        <generator class="native"/>
    </id>
    <set name="orderItems" inverse="true">
        <key column="ORDER_ID"/>
        <one-to-many class="OrderItem"/>
    </set>
</class>
```

Bidirectional

- To make the association bidirection, OrderItem would also need appropriate getters/setters and this mapping



```
<class name="OrderItem">
  <id name="id" column="ORDER_ITEM_ID">
      <generator class="native"/>
  </id>
  <many-to-one name="order" column="ORDER_ID"
  class="Order"/>
</class>
```

Other types of Associations

- Hibernate provides for the other types of multiplicity
  - One to One
  - Many to Many
- It also supports some unique mappings
  - Component mapping (Customer/Address where both objects are in the same row in DB)
  - Collection of Components
  - Collection of "value types" like date objects, String, etc
  - Polymorphic associations (association to a payment might actually be a check or credit card payment object)

Inheritance

- Hibernate also handles inheritance associations
- In fact, it supports three strategies with regard to inheritance mapping
  - Table per concrete class
  - Table per subclass
  - Table per class hierarchy
- Let’s look at an example inheritance tree and examine the options

Table per concrete class

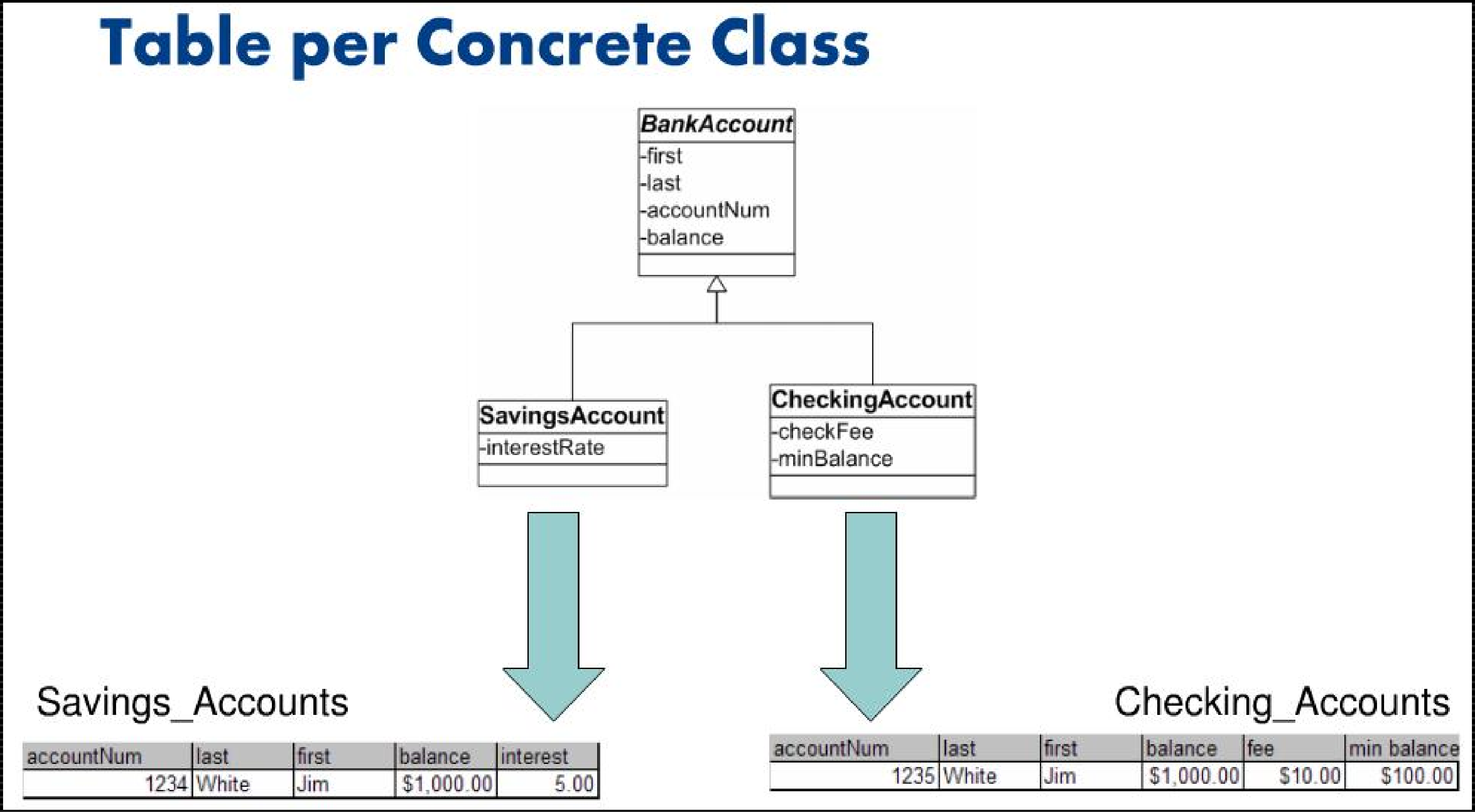


Table per Concrete Class

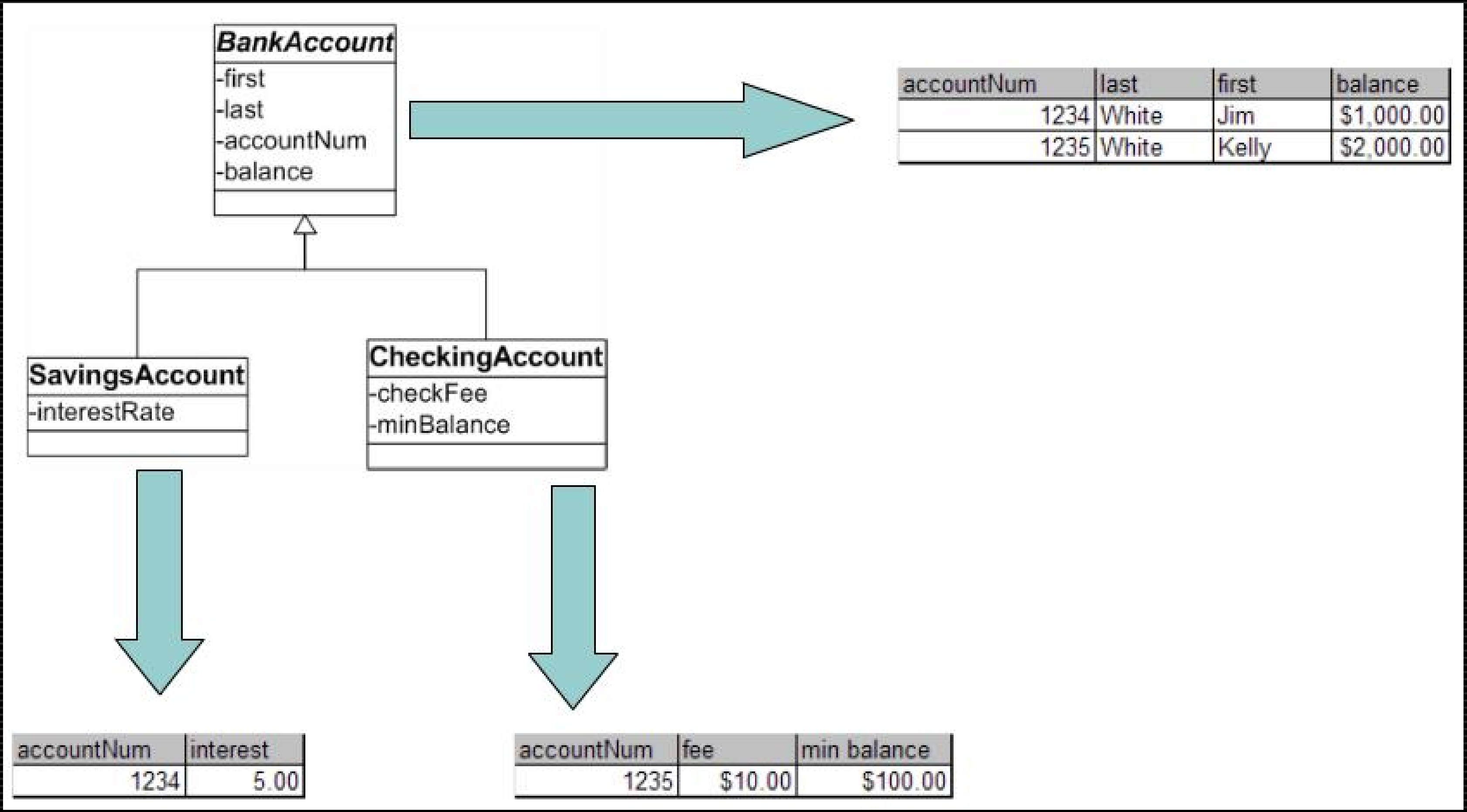
Savings_Accounts

accountNum	last	first	balance	interest
1234	White	Jim	$1,000.00	5.00

Checking_Accounts

accountNum	last	first	balance	fee	min balance
1235	White	Jim	$1,000.00	$10.00	$100.00

Table per Subclass

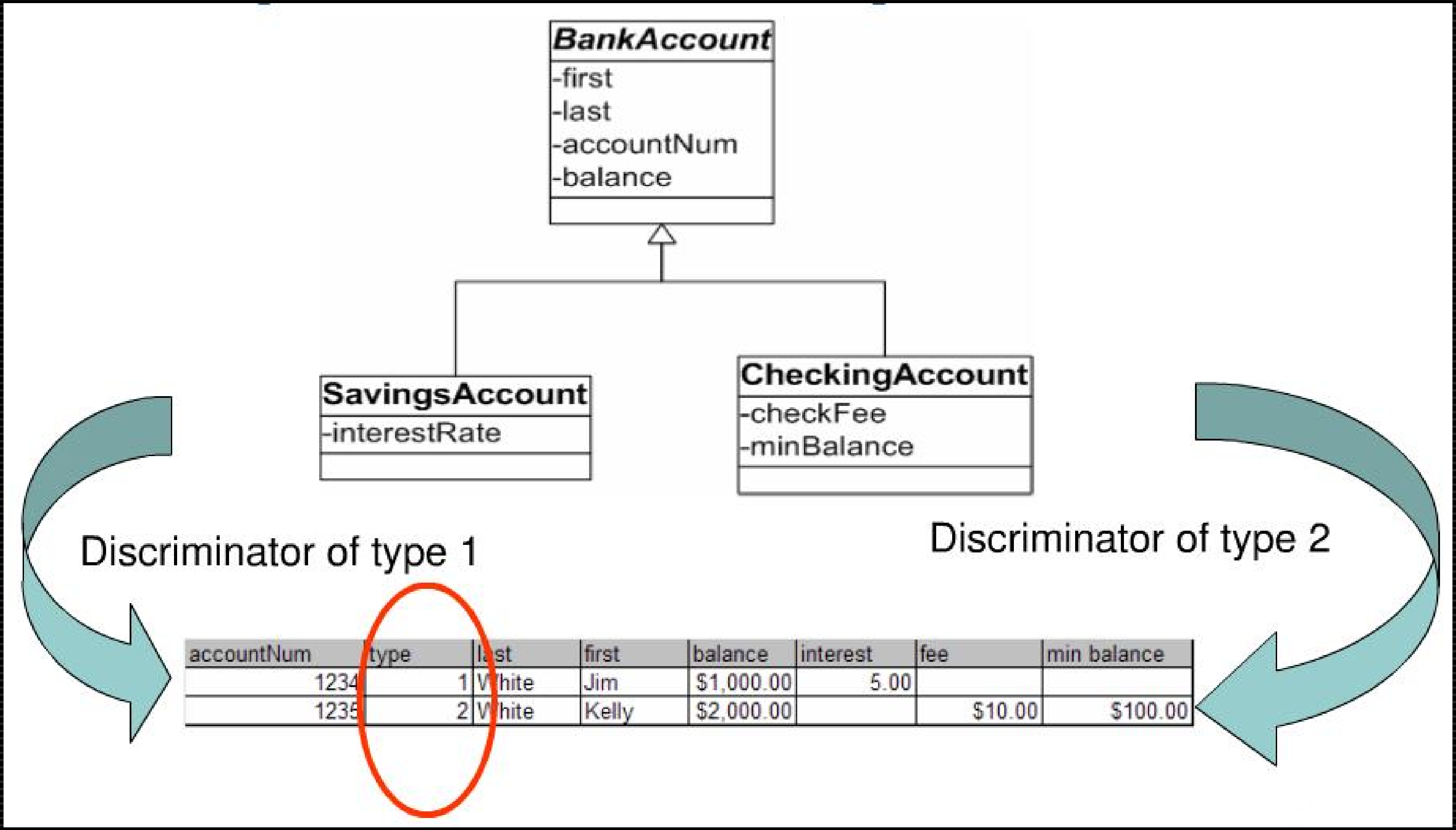


accountNum	last	first	balance
1234	White	Jim	$1,000.00
1235	White	Kelly	$2,000.00

accountNum	interest
1234	5.00

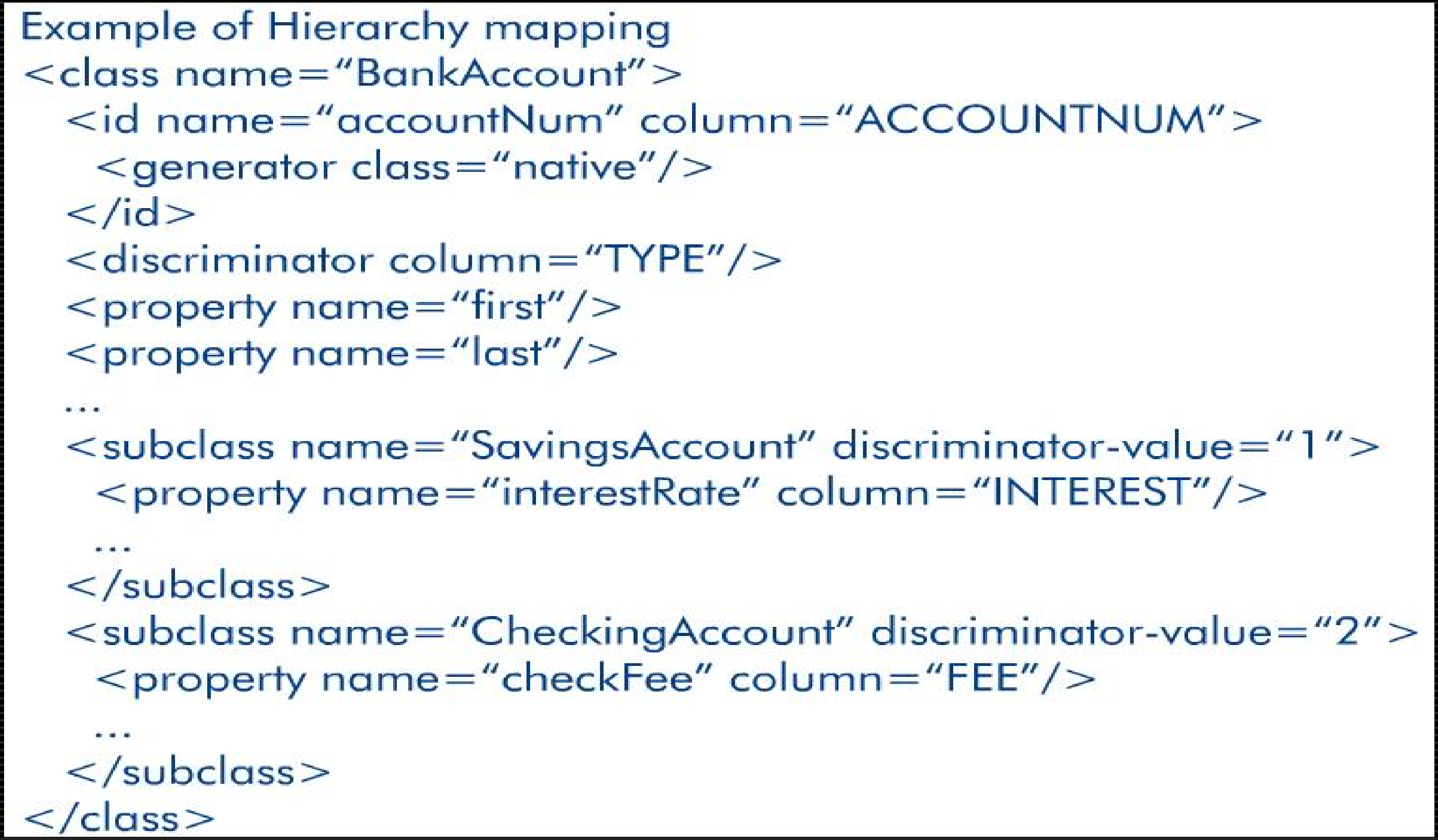
accountNum	fee	min balance
1235	$10.00	$100.00

Table per Class Hierarchy



accountNum	type	last	first	balance	interest	fee	min balance
1234	1	White	Jim	$1,000.00	5.00		
1235	2	White	Kelly	$2,000.00		$10.00	$100.00

Inheritance in the Mapping



Example of Hierarchy mapping

```
<class name="BankAccount">
   <id name="accountNum" column="ACCOUNTNUM">
     <generator class="native"/>
   </id>
   <discriminator column="TYPE"/>
   <property name="first"/>
   <property name="last"/>
   ...
   <subclass name="SavingsAccount" discriminator-value="1">
     <property name="interestRate" column="INTEREST"/>
     ...
   </subclass>
   <subclass name="CheckingAccount" discriminator-value="2">
     <property name="checkFee" column="FEE"/>
     ...
   </subclass>
</class>
```

Queries

- Hibernate supports several ways of getting objects recreated from data in the database
  - Hibernate also has non-object query capabilities for things like reports
- Single object retrieval has already been demonstrated
  - session.get(Customer.class, id)
- Standard SQL can be used when absolutely necessary
- The query capabilities are too numerous to show here, but we can give you a few examples of the options

HQL

- Hibernate Query Language (HQL) is an object oriented query language for performing queries
  - ANSI SQL based
  - Provides parameter binding like JDBC
- Examples

```
Query q = session.createQuery(“from Customer”);
List list = q.list(); // get all customers in a list
Customer tom = (Customer)
    session.createQuery( “from Customer c where
    c.id=1”).uniqueResult(); // single customer
```

Criteria Queries

- As an alternative to HQL, Hibernate offers criteria queries
  - More object oriented in approach
  - Queries can be partially checked in compile time versus runtime
- Examples

```
List list =
    session.createCriteria
    ( Customer.class ).add( Restrictions.eq("gender", 
    "F")).list(); // retrieve all female customers

List list2 =
    session.createCriteria
    ( Customer.class ).add(Restrictions.like("name", 
    "D%")).addOrder(Order.asc("dateOfBirth")).list(); // all 
    customers ordered by dateOfBirth whose name starts 
    with D
```

Fetching Strategies

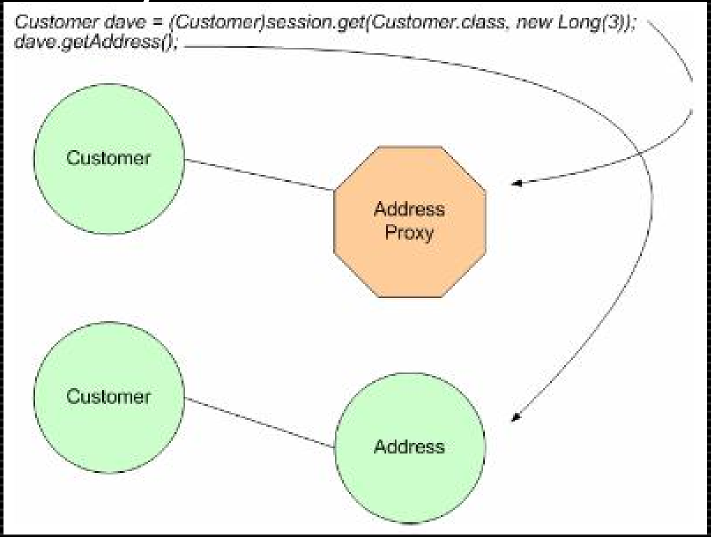
- Along with fetching objects there is a problem when objects are related to other objects
  - When you fetch Customer, should you also fetch associated, Address, Order, OrderItem etc. objects?
  - If not, what happens when you do something like the following?

```
Customer date = (Customer) session.get(Customer.class, new Long(3));
dave.getAddress();
```

  - Object graphs can be quite large and complex. How much and when should objects be retrieved?

Proxy Objects

- Hibernate provides for specifying both lazy and eager fetching
  - When lazy fetching is specified, it provides objects as stand-ins for associated objects
  - Proxy objects get replaced by the framework when a request hits a proxy.



```
Customer dave = (Customer)session.get(Customer.class, new Long(3));
dave.getAddress();
```

Flexibility and Extendibility

- Hibernate can be modified or extended
  - If you don’t like the way Hibernate handles a part of your persistence or you need capabilities it doesn’t offer, you can modify its functionality
  - Of course, it is open source, but there are also a number of extension points
- Extension points include
  - Dialects for different databases
  - Custom Mapping Types
  - Identifier Generator
  - Cache & CacheProvider
  - Transaction & TransactionFactory
  - PropertyAccessor
  - ProxyFactory
  - ConnectionProvider

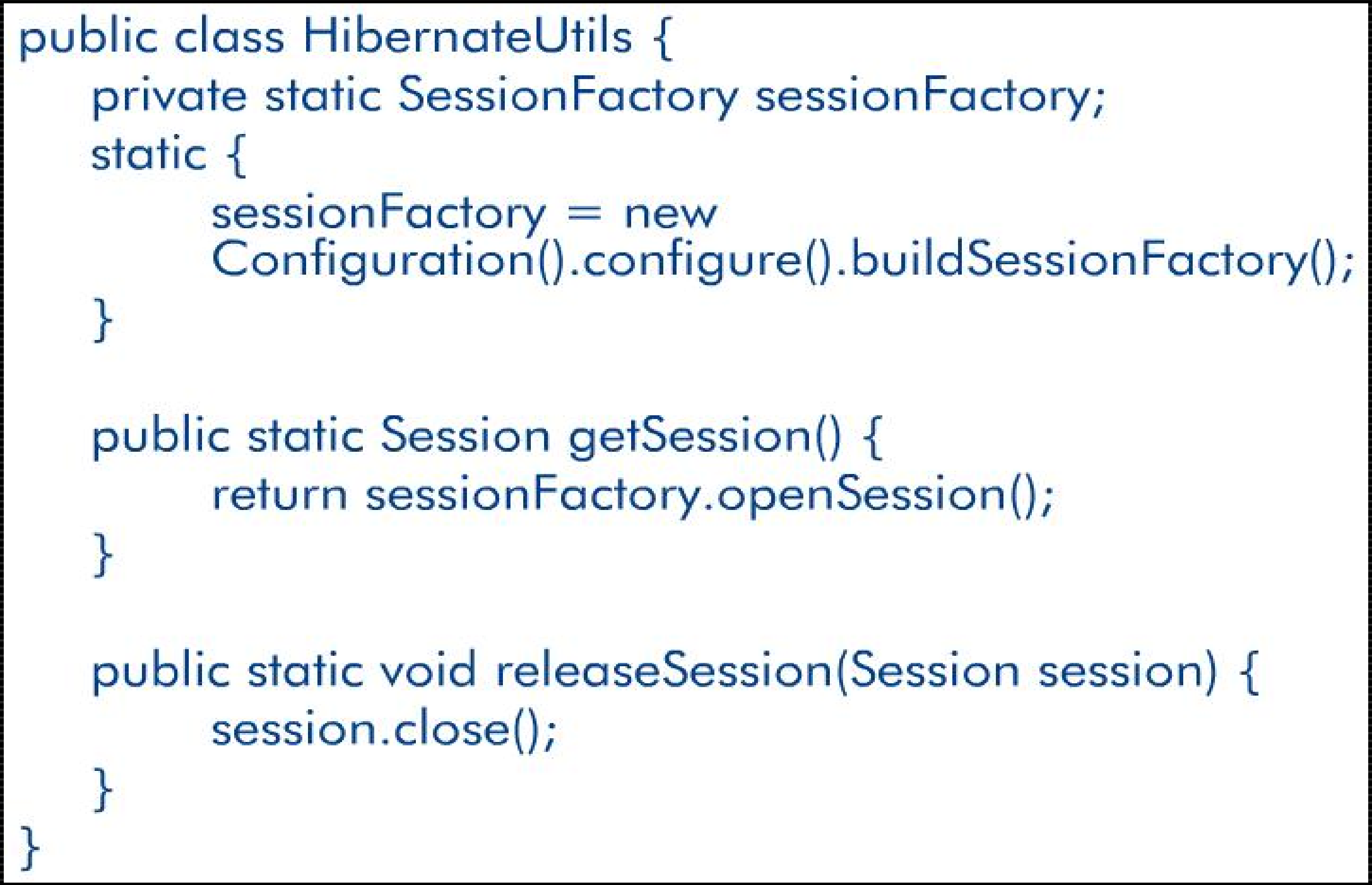
Good Design

- As with any technology, learning Hibernate is not hard
  - Using it effectively takes time and practice
  - There are a number of design patterns and tips
  - Some are actually provided with the hibernate documentation
  - Checkout Hibernate Tips/Tricks FAQ
- More background is needed to discuss some of the patterns and tips/tricks
  - Some make good sense no matter what ORM tool is used
  - A few are apparent even with the knowledge gained today

Layered Design Pattern

- Hibernate provides the means to perform persistence without code intrusion
- However, this does not negate the need for appropriate data access layers
- A common pattern applied to Hibernate applications is the separate layers and provide appropriate interfacing
- Allows Hibernate or other ORM to be more easily replaced or extended
- Provides appropriate separation of concerns

Hibernate Utils



```
public class HibernateUtils {
    private static SessionFactory sessionFactory;
    static {
            sessionFactory = new
            Configuration().configure().buildSessionFactory();
    }

    public static Session getSession() {
            return sessionFactory.openSession();
    }

    public static void releaseSession(Session session) {
            session.close();
    }
}
```

Hibernate Tips/Tricks

- Get a SessionFactory early and keep it
- Get Session and Transaction close to the unit of work and drop them
- Load Lazy by default (for individual objects)
- Use HQL or Criteria with eager fetching when subject to N + 1 fetches
- Hibernate offers two levels of Cache. Take care when configuring either
- Bags can provide horrible Collection performance
- If properties are only read but never updated, mark them appropriately

Resources

- www.hibernate.org
- http://theserverside.com
- Books
  - Hibernate in Action (Manning)
  - Hibernate a Developer’s Notebook (O’Reilly)
  - Java Persistence with Hibernate (Manning)
  - Hibernate: A J2EE Developer’s Guide (Addison Wesley)