4

Component and Inheritance Mapping

Objectives

- At the end of this lesson, you will be able to:

  - Understand the difference between Components and Entities
  - Analyze some uses of Components, and their mapping
  - Look at different strategies and implementations for realizing inheritance

Component

- **Refers to the UML modeling term of composition**
- **Does not exist on its own; dependent on a parent object**
  - Does not have a table or identifier in the database
  - Only associated to a single parent class
- **Commonly referred to as a "*has a*" relationship**
  - An **AccountOwner** *has an* **Address**

Entity

- 1st class citizen -- lives on its own
- Has its own table and identifier
- Can be made up of multiple components
- Can be related/associated to other entities

Entity/Component Example

'AccountOwner' is an Entity

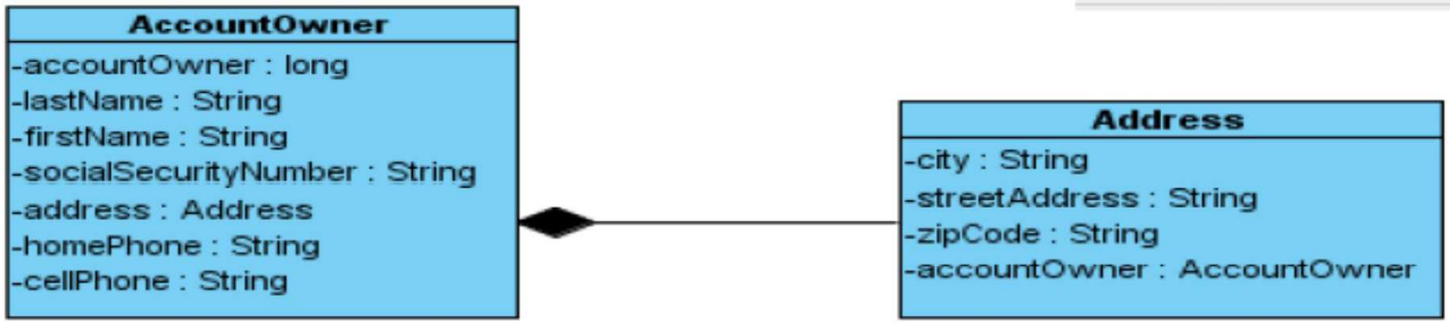
- Can lives on its own
- Has its own id

'Address' is a component of 'AccountOwner'

- Non-existent without AccountOwner
- No id of its own.

ACCOUNT_OWNER

Column Name	Data Type	Nullable
ACCOUNT_OWNER_ID	NUMBER	No
LAST_NAME	VARCHAR2(50)	No
FIRST_NAME	VARCHAR2(50)	No
SOCIAL_SECURITY_NUMBER	VARCHAR2(50)	No
STREET_ADDRESS	VARCHAR2(50)	No
CITY	VARCHAR2(50)	No
STATE	VARCHAR2(50)	No
ZIP_CODE	VARCHAR2(10)	No
HOME_PHONE	VARCHAR2(20)	Yes
CELL_PHONE	VARCHAR2(20)	Yes



Using Components

1. **Determine your domain model**
   - Which objects are best suited to be a component?
2. **Create your database tables**
   - Model your components accordingly within entity tables
3. **Create your Java classes**
   - One for each entity, and one for each component
   - Setup your components within the entity classes
4. **Write the Hibernate mapping file for the entity, using the embedded component tag within it to identify the component class.**

Mapping a Component

```
<class name="courses.hibernate.vo.AccountOwner"
       table="ACCOUNT_OWNER">
  <id name="accountOwnerId" column="ACCOUNT_OWNER_ID">
    <generator class="native" />
  </id>
  <property name="lastName" column="LAST_NAME" type="string" />
  <property name="firstName" column="FIRST_NAME" type="string" />
  <property name="socialSecurityNumber"
            column="SOCIAL_SECURITY_NUMBER" type="string" />
  <component name="address" class="courses.hibernate.vo.Address">
    <parent name="accountOwner"/>
    <property name="streetAddress" column="STREET_ADDRESS"
              type="string" />
    <property name="city" column="CITY" type="string" />
    <property name="state" column="STATE" type="string" />
    <property name="zipCode" column="ZIP_CODE" type="string" />
  </component>
  <property name="homePhone" column="HOME_PHONE" type="string" />
  <property name="cellPhone" column="CELL_PHONE" type="string" />
</class>
```

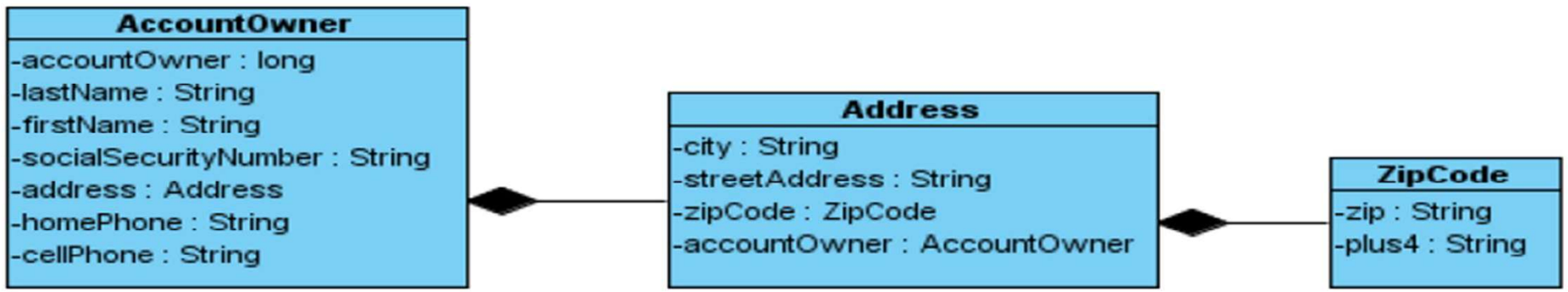
Nested Components

Component 'Address' has *nested* component 'ZipCode'

- Example: 12222-1234

ACCOUNT_OWNER

Column Name	Data Type	Nullable
ACCOUNT_OWNER_ID	NUMBER	No
LAST_NAME	VARCHAR2(50)	No
FIRST_NAME	VARCHAR2(50)	No
SOCIAL_SECURITY_NUMBER	VARCHAR2(50)	No
STREET_ADDRESS	VARCHAR2(50)	No
CITY	VARCHAR2(50)	No
STATE	VARCHAR2(50)	No
ZIP_CODE	VARCHAR2(5)	No
ZIP_PLUS_FOUR	VARCHAR2(4)	Yes
HOME_PHONE	VARCHAR2(50)	Yes
CELL_PHONE	VARCHAR2(50)	Yes



Mapping Nested Components

```
<class name="courses.hibernate.vo.AccountOwner"
       table="ACCOUNT_OWNER">
  ...

  <component name="address" class="courses.hibernate.vo.Address">
    <parent name="accountOwner"/>
      <property name="streetAddress" column="STREET_ADDRESS"
                type="string" />
      <property name="city" column="CITY" type="string" />
      <property name="state" column="STATE" type="string" />
      <component name="zipCode"
                 class="courses.hibernate.vo.ZipCode">
        <property name="zip" column="ZIP_CODE" type="string" />
        <property name="plus4" column="ZIP_PLUS_FOUR"
                  type="string" />
      </component>
  </component>

  ...
  </class>
```

Collection of Components

```
<class name="courses.hibernate.vo.Account" table="ACCOUNT">
    ...
  <set name="accountOwnerAddresses" table="ACCOUNT_ACCOUNT_OWNER">
    <key column="ACCOUNT_ID" />
    <composite-element class="courses.hibernate.vo.Address">
      <property name="streetAddress" type="string"
        formula="(SELECT AO.STREET_ADDRESS FROM ACCOUNT_OWNER AO
                  WHERE AO.ACCOUNT_OWNER_ID = ACCOUNT_OWNER_ID)"/>

      <property name="city" type="string"
          formula="(SELECT AO.CITY FROM ACCOUNT_OWNER AO WHERE
                    AO.ACCOUNT_OWNER_ID = ACCOUNT_OWNER_ID)" />

      <property name="state" type="string"
          formula="(SELECT AO.STATE FROM ACCOUNT_OWNER AO WHERE
                    AO.ACCOUNT_OWNER_ID = ACCOUNT_OWNER_ID)" />
    </composite-element>
  </set>
```

Component as Entity ID

- **Done with or without a separate Java class**
  - If using Java class
    - Must implement Comparable and Serializable
    - Must define ‘class’ attribute in mapping file

Example: EBill uses EBillId component as primary key

- EBillId component class contains attributes required to uniquely identiry the EBill

EBillId Class

```
public class EBillId implements
   Comparable<EBillId>, Serializable {
   private long accountId;
   private long ebillerId;
   private Date dueDate;
   ...
   // getters and setters
   ...
}
```

Example: EBill uses EBillId component as primary key

- In EBill entity class, EBillId values are set as appropriate setters are called.

EBill Class

```
public class EBill {
  private EBillId ebillId = new EBillId();
  private EBiller ebiller;
        ...
  protected void setEbiller(EBiller ebiller) {
    this.ebiller = ebiller;
    ebillId.setEbillerId(ebiller.getEbillerId());
    ...
  }
  ...
}
```

Example: EBill uses EBillId component as primary key

EBill Mapping File

```
<class name="courses.hibernate.vo.EBill" table="EBILL">
  <composite-id name="ebillId"
                class="courses.hibernate.vo.EBillId">
    <key-property name="accountId" column="ACCOUNT_ID"
                  type="long"/>
    <key-property name="ebillerId" column="EBILLER_ID"
                  type="long"/>
    <key-property name="dueDate" column="DUE_DATE"
                  type="timestamp"/>
  </composite-id>
  ...
  <!-- Must have insert="false" update="false" because ids for the objects are
   part of the composite key. The relationships are managed via the composite
   key elements rather than the M:1 relationship. -->
  <many-to-one name="ebiller" column="EBILLER_ID"
           class="courses.hibernate.vo.EBiller" access="field"
           insert="false" update="false" />
  ...
</class>
```

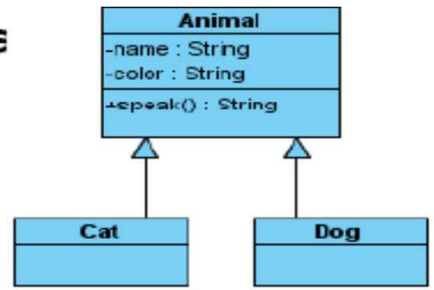
Inheritance

- **Allows for logical affiliation of classes with common state and/or behavior**
- **Commonly referred to as an "*is a*" relationship**
  - A **Cat** *is a* **Animal**
  - A **Dog** *is a* **Animal**
- **Polymorphism**
  - Dynamic realization of subclass behavior while treating the object as an instance of its superclass

Polymorphism

```
public class Animal {
   public String speak() {
      return "Generic Hello";
   }
}
public class Cat extends Anima
   public String speak() {
      return "Meow";
   }
}
public class Dog extends Animal {
   public String speak() {
      return "Woof";
   }

  }
```



Inheritance Realization

- **Easy to do in an object oriented language**
  - Java: Use ‘extends’ or ‘implements’ keyword
- **Not so easy to do in a relational database**
  - Tables do not ‘extend’ from each other
  - Part of the ‘impedance mismatch’ problem
- **How do we get around this?**
  - Four approaches through Hibernate
    - Hibernate implicit polymorphism
    - Table-per-concrete class
    - Table-per-class-hierarchy
    - Table-per-subclass

Modeling Inheritance

1. Determine your domain model

- What objects have hierarchical relationships?

2. Choose your inheritance strategy

- Hibernate implicit polymorphism
- Table-per-concrete class
- Table-per-class-hierarchy
- Table-per-subclass

3. Create your database tables based on the chosen strategy

4. Code your Java objects, using ‘extends’ or ‘implements’

5. Write your Hibernate mapping file using the appropriate subclass tags

Implicit Polymorphism

- **Database**
  - *One* database table *per* concrete class
- **Hibernate Mapping Files**
  - *Separate mapping files for each inherited class*
    - Like normal, including any inherited properties
- **Default Behavior**
  - Out of the box Hibernate will automatically recognize any Java inheritance associations

Implicit Polymorphism

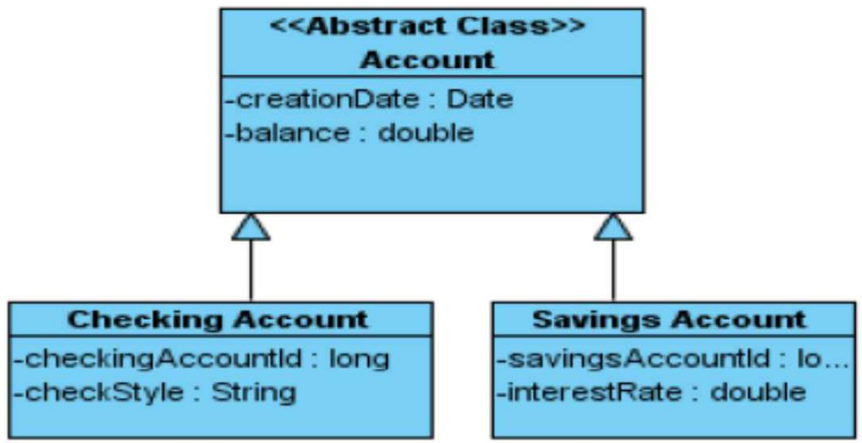
- One table per concrete class

CHECKING_ACCOUNT

Column Name	Data Type	Nullable
CHECKING_ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
CHECK_STYLE	VARCHAR2(50)	No

SAVINGS_ACCOUNT

Column Name	Data Type	Nullable
SAVINGS_ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
INTEREST_RATE	NUMBER(10,2)	No



- CheckingAccount mapping file

```
<class name="courses.hibernate.vo.CheckingAccount"
       table="CHECKING_ACCOUNT">

  <id name="checkingAccountId" column="CHECKING_ACCOUNT_ID">
    <generator class="native"/>
  </id>

  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"    update="false"/>

  <property name="balance" column="BALANCE" type="double"/>

  <property name="checkStyle" column="CHECK_STYLE"
            type="string"/>
</class>
```

- SavingsAccount mapping file

```
<class name="courses.hibernate.vo.SavingsAccount"
       table="SAVINGS_ACCOUNT">

  <id name="savingsAccountId" column="SAVINGS_ACCOUNT_ID">
    <generator class="native"/>
  </id>

  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"    update="false"/>

  <property name="balance" column="BALANCE" type="double"/>

  <property name="interestRate" column="INTEREST_RATE"
            type="double"/>
</class>
```

Implicit Polymorphism

- **Advantages**
  - Get it for free. Hibernate automatically scans classes on startup (including inheritance); No additional configuration/mapping needed
  - Not a lot of nullable columns *(good for integrity)*
  - Queries against individual types are fast and simple

Implicit Polymorphism

- **Disadvantages**
  - Makes handling relationships difficult

Table-per-concrete class

- **Database**
  - *One* database table *per* concrete class
- **Hibernate Mapping**
  - *Single mapping file*
    - Based on superclass
  - Includes '**union-subclass**' definitions for inherited classes

Table-per-concrete class

- One table per concrete class

CHECKING_ACCOUNT

Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
CHECK_STYLE	VARCHAR2(50)	No

SAVINGS_ACCOUNT

Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
INTEREST_RATE	NUMBER(10,2)	No

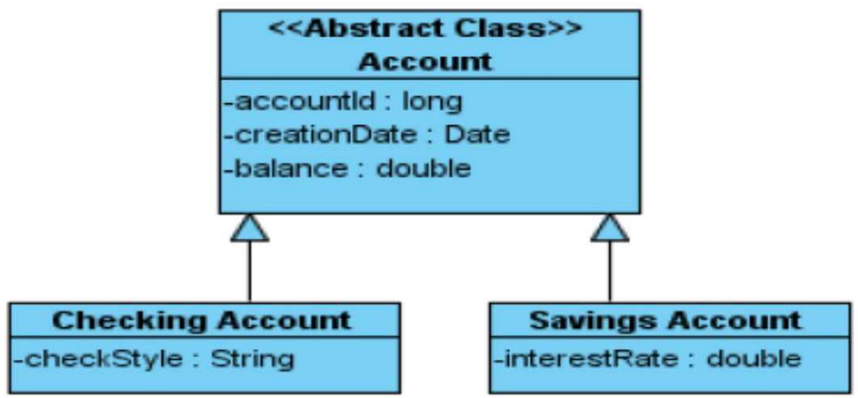


Table-per-concrete class

Account Mapping File

```
<class name="Account" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long">
    <generator class="native"/>
  </id>

  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"/>
  <property name="balance" column="BALANCE"
            type="double"/>

  <union-subclass name="courses.hibernate.vo.SavingsAccount"
                  table="SAVINGS_ACCOUNT">
    <property name="interestRate"
              column="INTEREST_RATE" type="double"/>
  </union-subclass>
  <union-subclass name="courses.hibernate.vo.CheckingAccount"
                  table="CHECKING_ACCOUNT">
    <property name="checkStyle" column="CHECK_STYLE"
              type="string"/>
  </union-subclass>
</class>
```

Table-per-concrete class

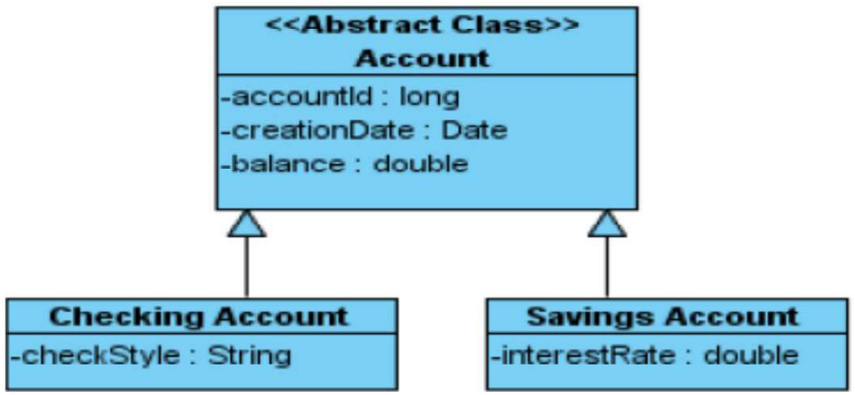
- **Advantages**
  - Shared mapping of common elements
    - Shared database id
  - Not a lot of nullable columns *(good for integrity)*
  - Queries against individual types are fast and simple
  - Less SQL statements generated with use of ‘Union’ for polymorphic queries
- **Disadvantages**
  - Still have difficulty with relationships
    - Foreign keying to two tables not possible

Table-per-class-hierarchy

- **Database**
  - *One* database table for *all* subclasses
  - Denormalized table has columns for all attributes
- **Hibernate Mapping**
  - *Single mapping file* still based on superclass
  - Includes ‘subclass’ definitions for inherited classes
  - Use ‘discriminator’ column/field to identity concrete type

Table-per-class-hierarchy

- One table for all inherited classes



ACCOUNT

Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No
ACCOUNT_TYPE	VARCHAR2(1)	No
CHECK_STYLE	VARCHAR2(50)	Yes
INTEREST_RATE	NUMBER(10,2)	Yes

ACCOUNT_ID	CREATION_DATE	BALANCE	ACCOUNT_TYPE	CHECK_STYLE	INTEREST_RATE
1	17-AUG-08 06.03.27.000000 PM	1000	C	Sea Creatures	-
2	09-AUG-08 06.03.45.000000 PM	6000	C	Angels	-
3	09-SEP-08 06.04.24.000000 PM	12000	S	-	.25
4	09-SEP-08 06.04.53.000000 PM	8000	S	-	4.2

Table-per-class-hierarchy

Account Mapping File

```
<class name="Account" table="ACCOUNT" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long"
    <generator="native"/>
  </id>
  <discriminator column="ACCOUNT_TYPE" type="string"/>
  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"/>
  <property name="balance" column="BALANCE" type="double"/:

  <subclass name="courses.hibernate.vo.SavingsAccount"
            discriminator-value="S">
    <property name="interestRate" column="INTEREST_RATE"/>
  </subclass>

  <subclass name="courses.hibernate.vo.CheckingAccount"
            discriminator-value="C">
    <property name="checkStyle" column="CHECK_STYLE"/>
  </subclass>
</class>
```

Table-per-class-hierarchy

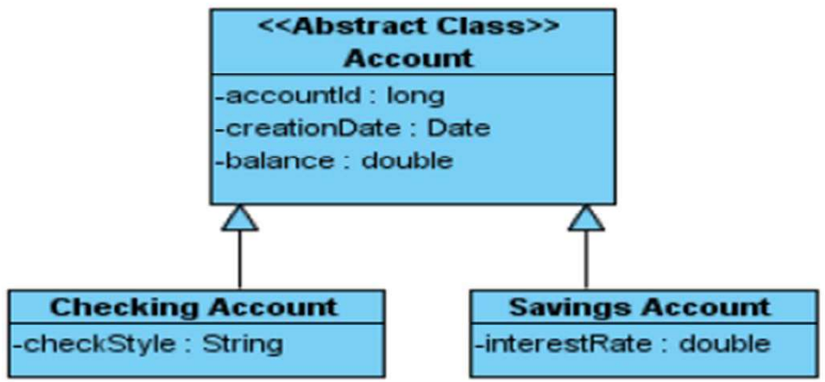
- **Advantages**
  - Simple
  - Fast reads/writes, even across types
- **Disadvantages**
  - Lots of nullable columns
    - Possible data integrity concern
  - Denormalized table generally considered bad database design

Table-per-subclass

- **Database**
  - *One* database table for the superclass ***AND*** one *per* subclass
    - Shared columns in superclass table
    - Subclass tables have their object-specific columns
- **Hibernate Mapping File**
  - *Single mapping file* based on the superclass
  - Includes ‘joined-subclass’ definitions for inherited classes

Table-per-subclass

- **Every class that has persistent properties has its own table**
  - Each table contains a primary key, and non-inherited properties
  - Inheritance is realized through foreign keys



Column Name	Data Type	Nullable
ACCOUNT_ID	NUMBER	No
CREATION_DATE	TIMESTAMP(6)	No
BALANCE	NUMBER(10,2)	No

Column Name	Data Type	Nullable
CHECKING_ACCOUNT_ID	NUMBER	No
CHECK_STYLE	VARCHAR2(50)	No

Column Name	Data Type	Nullable
SAVINGS_ACCOUNT_ID	NUMBER	No
INTEREST_RATE	NUMBER(10,2)	No

Table-per-subclass

Account Mapping File

```
<class name="Account" table="ACCOUNT" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long">
    <generator class="native"/>
  </id>
  <property name="creationDate" column="CREATION_DATE"
            type="timestamp"/>
  <property name="balance" column="BALANCE"
            type="double"/>
  <joined-subclass name="courses.hibernate.vo.SavingsAccount"
                   table="SAVINGS_ACCOUNT">
    <key column="SAVINGS_ACCOUNT_ID"/>
    <property name="interestRate" column="INTEREST_RATE"
              type="double"/>
  </joined-subclass>
  <joined-subclass name="courses.hibernate.vo.CheckingAccount'
                   table="CHECKING_ACCOUNT">
    <key column="CHECKING_ACCOUNT_ID"/>
    <property name="checkStyle" column="CHECK_STYLE"
              type="string"/>
  </joined-subclass>
</class>
```

Table-per-subclass

- **Advantages**
  - Normalized schema
    - Schema evolution and integrity are straight forward
  - Reduced number of SQL statements produced
    - Hibernate uses inner joins for subclass queries.
- **Disadvantages**
  - Can have poor performance for complex systems

When to use Which

- Leave implicit polymorphism for queries against interfaces *(based on behavior, not different attributes)*
- If you rarely require polymorphic queries, lean towards table-per-concrete-class.
- If polymorphic behavior is required, AND subclasses have only a few distinct properties, try table-perclass-hierarchy
- If polymorphic AND many distinct properties, look at table-per-subclass or table-per-concrete-class, weighing the cost of joins versus unions

Summary

- In this lesson, we have learned
  - Learnt how Components are different from Entities
    - Enties have their own IDs
    - Component are dependant on Entities
    - Java: Object references
  - Spoke about the different methods of modeling inheritance relationships through Java/Database/Hibernate, and where each one is best suited.
    - Hibernate implicit polymorphism
    - Table-per-concrete class
    - Table-per-class-hierarchy
    - Table-per-subclass