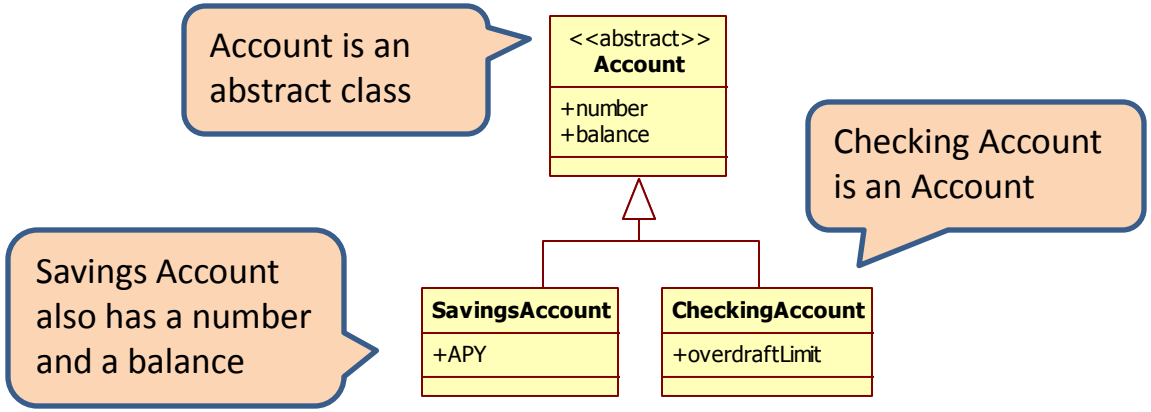
Module 05: Inheritance Mapping

CS544: Enterprise Architecture

Inheritance

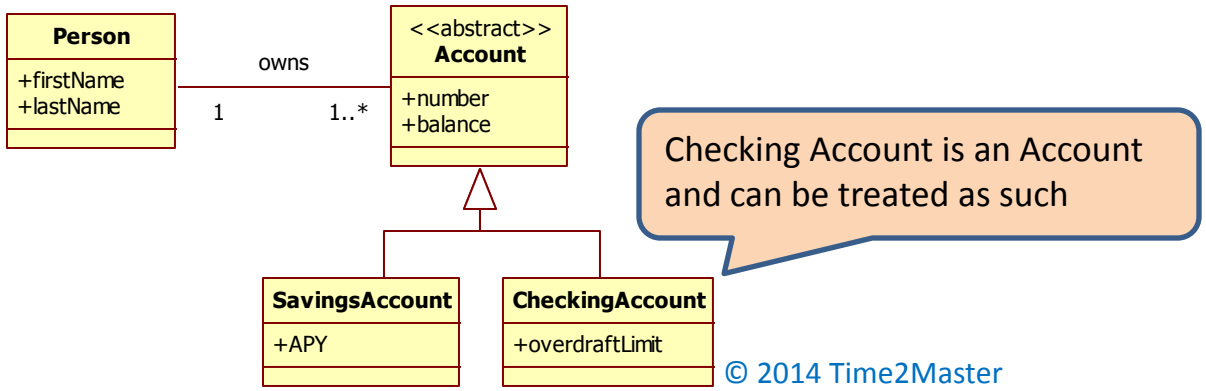
- Using inheritance a class can extend another class
  - Thereby inheriting all its properties and method
  - Often referred to as an ‘is-a’ relationship



- Relational databases don’t have ‘is-a’ relationships
  - There are 3 different ways to emulate inheritance

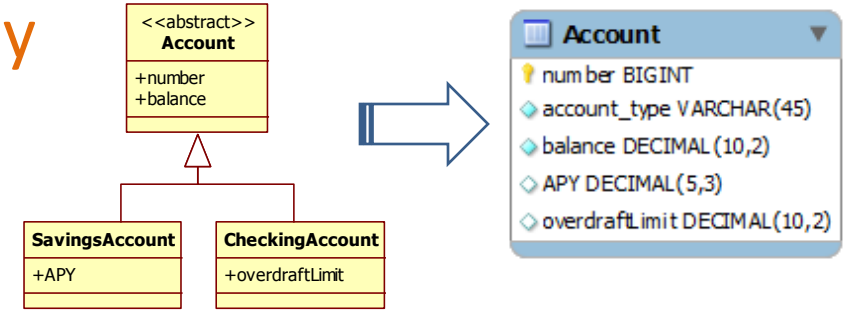
Polymorphism

- Polymorphism is the ability of a subtype to appear and behave like its supertype
- This enables Person to have a list of Account references, which may hold SavingsAccounts and CheckingAccounts.
- A polymorphic query is a query for all objects in a hierarchy, independent of their subtype

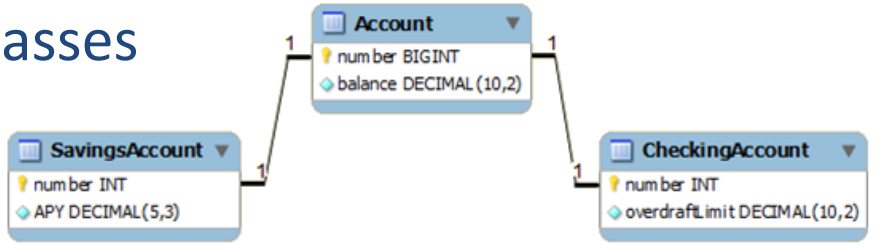


Three ways to map

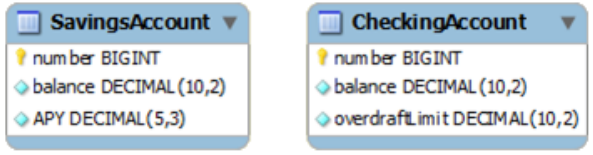
- You can map inheritance in one of three ways:
  - Single Table per Hierarchy
    - De-normalized schema
    - Fast polymorphic queries



  - Joined Tables
    - Normalized & similar to classes
    - Slower queries



  - Table per Concrete Class
    - Uses UNION instead of JOIN
    - All needed columns in each table

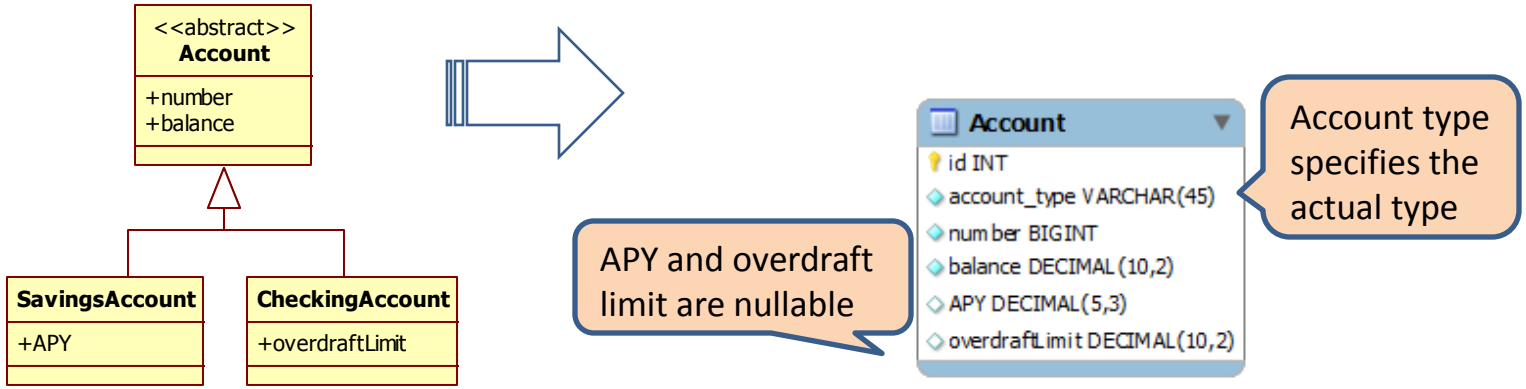


Inheritance Mapping

SINGLE TABLE PER HIERARCHY

Single Table Implementation

- Single Table per Hierarchy uses one big table
  - Discriminator column specifies actual type
  - Sub class properties added as nullable columns



Single Table in Action

ACCOUNT_TYPE	NUMBER	BALANCE	OVERDRAFTLIMIT	APY
checking	1	500	200	
savings	2	100		2.3
checking	3	23.5	0	

APY is null for checking accounts, overdraft limit is null for savings

+ Simple, Easy to implement
+ Good performance on all queries, polymorphic and non polymorphic

– Nullable columns / de-normalized schema
– Table may have to contain lots of columns

SQL for Single Table

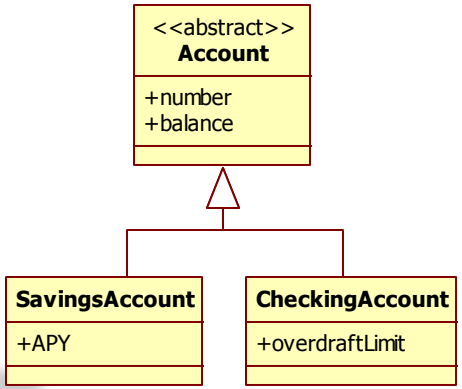
```
select
        account0_.number as number0_,
        account0_.balance as balance0_,
        account0_.owner_id as owner6_0_,
        account0_.overdraftLimit as overdraf4_0_,
        account0_.APY as APY0_,
        account0_.account_type as account1_0_
    from
        Account account0_
```

Single Table Mapping

Specify the SINGLE_TABLE strategy

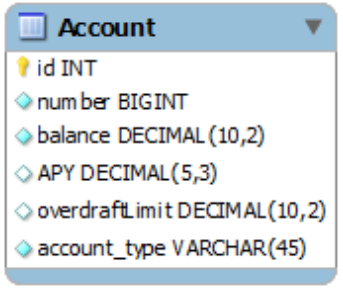
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="account_type",
    discriminatorType=DiscriminatorType.STRING
)
public abstract class Account
  @Id
  @GeneratedValue
  private long number;
  private double balance;

  ...
```

Optional annotation @DiscriminatorColumn




```
@Entity
@DiscriminatorValue("savings")
public class SavingsAccount extends Account {
  private double APY;

  ...
```

Specify discriminator value




```
@Entity
@DiscriminatorValue("checking")
public class CheckingAccount extends Account {
  private double overdraftLimit;

  ...
```

Specify discriminator value

XML

```
<hibernate-mapping package="single">
  <class name="Account" abstract="true">
    <id name="number">
      <generator class="native" />
    </id>

    <discriminator type="string" column="account_type" />

    <property name="balance" />

    <subclass name="SavingsAccount" discriminator-value="savings">
      <property name="APY" />
    </subclass>

    <subclass name="CheckingAccount" discriminator-value="checking">
      <property name="overdraftLimit" />
    </subclass>
  </class>
</hibernate-mapping>
```

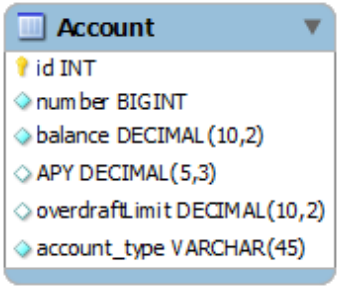
Abstract account class

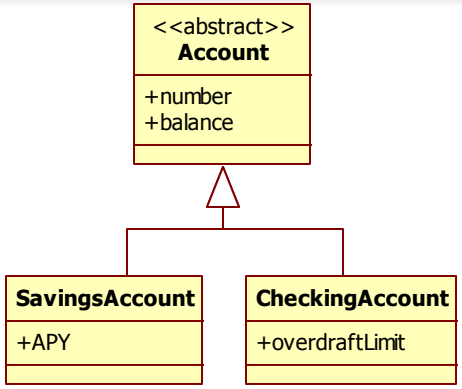
<discriminator> tag has to be after <id> and before <property>

Account properties

Subclass definition included inside superclass

Discriminator values optional



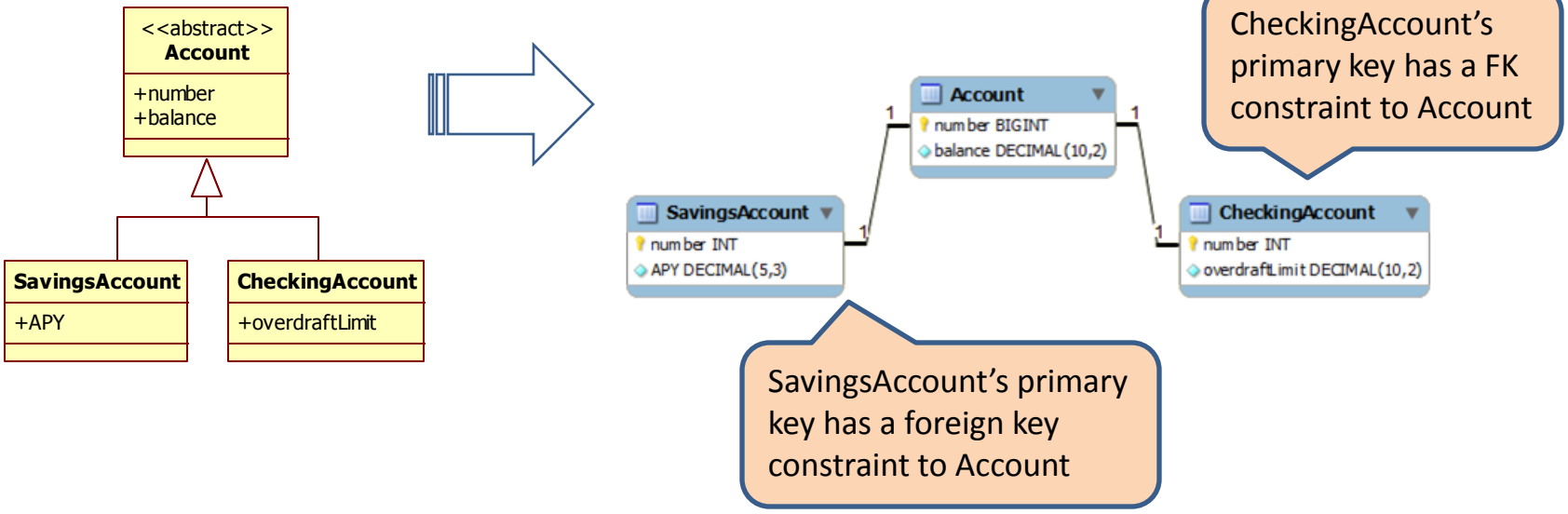


Inheritance Mapping

JOINED TABLES

Joined Tables Implementation

- The Joined Tables Strategy uses FK ‘has a’ relations to emulate ‘is a’ relations
  - Uses Foreign Key constraints on the Primary Keys
  - Queries use JOINs to included needed tables



Joined Tables in Action

Account Table

NUMBER	BALANCE
1	500
2	100
3	23.5

SavingsAccount

NUMBER	APY
2	2.3

CheckingAccount

NUMBER	OVERDRAFTLIMIT
1	200
3	0

+ Normalized Schema
+ Database view is similar to domain view
– Inserting or updating an entity results in multiple insert or update statements
– Necessary joins can give lower query performance

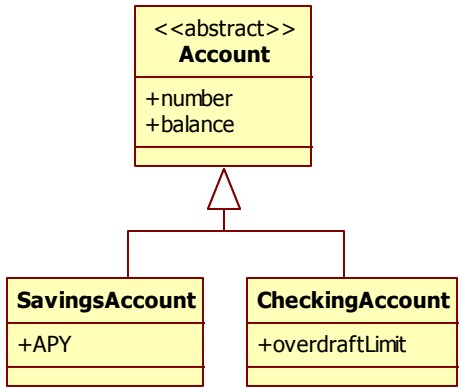
Joined – No Discriminator Value

```
select
        account0_.number as number0_,
        account0_.balance as balance0_,
        account0_.owner_id as owner3_0_,
        account0_1_.overdraftLimit as overdraf1_1_,
        account0_2_.APY as APY2_,
        case
            when account0_1_.number is not null then 1
            when account0_2_.number is not null then 2
            when account0_.number is not null then 0
        end as clazz_
    from
        Account account0_
    left outer join
        CheckingAccount account0_1_
            on account0_.number=account0_1_.number
    left outer join
        SavingsAccount account0_2_
            on account0_.number=account0_2_.number
```

Joined

Just specify the inheritance strategy, nothing else

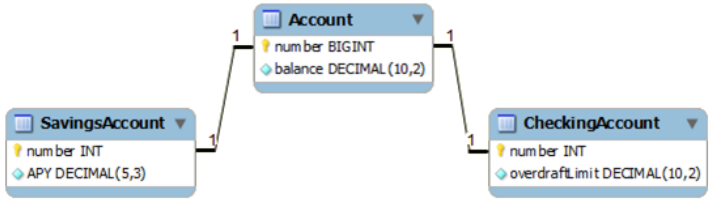
```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Account {
   @Id
   @GeneratedValue
   private long number;
   private double balance;

   ...
```




```
@Entity
public class SavingsAccount extends Account {
   private double APY;

   ...
```

Subclasses can be mapped as normal entity classes, but without identifiers

```
@Entity
public class CheckingAccount extends Account {
   private double overdraftLimit;

   ...
```



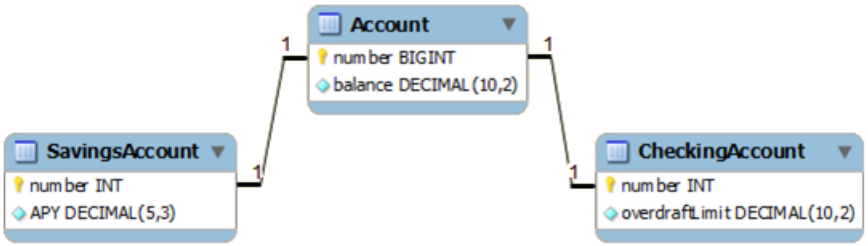
XML

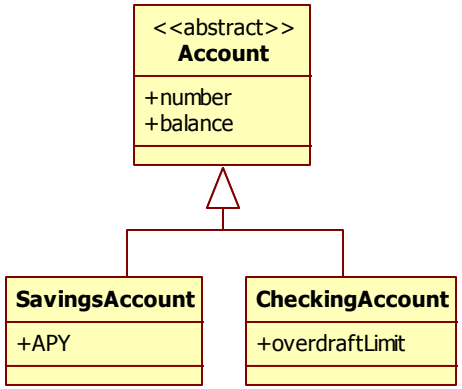
```
<hibernate-mapping package="single">
  <class name="Account" abstract="true">
    <id name="number">
      <generator class="native" />
    </id>

    <property name="balance" />

    <joined-subclass name="SavingsAccount">
      <key column="number" />
      <property name="APY" />
    </joined-subclass>

    <joined-subclass name="CheckingAccount">
      <key column="number" />
      <property name="overdraftLimit" />
    </joined-subclass>
</class>
</hibernate-mapping>
```

Abstract account class

Joined Subclasses are also specified inside the super class

Need an additional <key> tag to specify the PK / join column in the subclasses



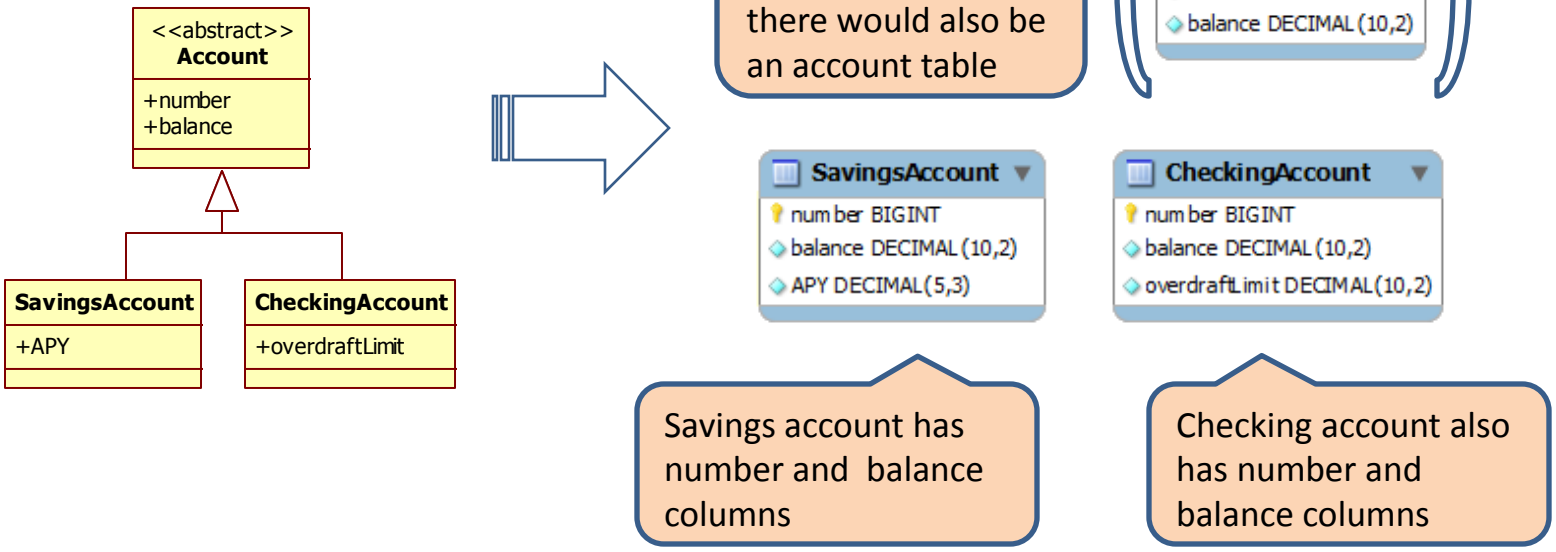


Inheritance Mapping

TABLE PER CONCRETE CLASS

Table per Concrete Class

- Table per Concrete Class uses a table for each concrete (non-abstract) class
  - Subclass tables include **all** superclass properties
  - Polymorphic queries use UNION operations



Concrete Class in Action

SavingsAccount

NUMBER	BALANCE	APY
2	100	2.3

CheckingAccount

NUMBER	BALANCE	OVERDRAFTLIMIT
1	500	200
3	23.5	0

+ Very efficient non-polymorphic queries
+ Decently efficient polymorphic queries

– Cannot use Identity column ID generation
– Polymorphic one too many associations only supported for bi-directional associations
– Subclass properties cannot use primitives
– JPA does not require its implementation (optional)

Table Per Concrete – No Discriminator Value

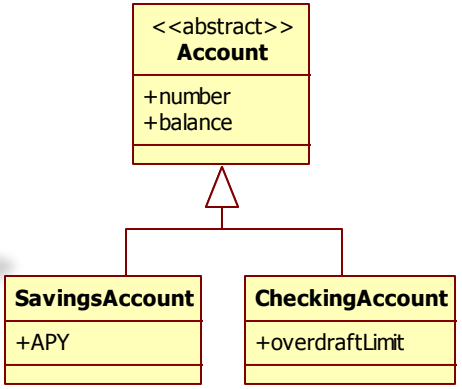
```
select
        account0_.number as number0_,
        account0_.balance as balance0_,
        account0_.owner_id as owner3_0_,
        account0_.overdraftLimit as overdraf1_1_,
        account0_.APY as APY2_,
        account0_.clazz_ as clazz_
    from
        ( select
            number,
            balance,
            owner_id,
            overdraftLimit,
            cast(null as int) as APY,
            1 as clazz_
        from
            CheckingAccount
        union
        all select
            number,
            balance,
            owner_id,
            cast(null as int) as overdraftLimit,
            APY,
            2 as clazz_
        from
            SavingsAccount
    ) account0_
```

Table per Class

Just specify the inheritance strategy, nothing else

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Account {
   @Id
   @GeneratedValue(strategy=GenerationType.TABLE)
   private long number;
   private double balance;

   ...
```

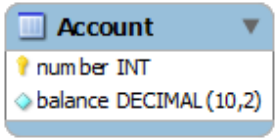
Id generation can not use identity column

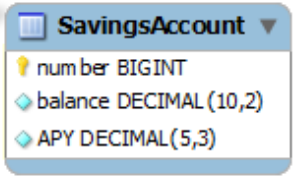


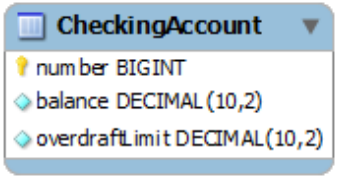
Normal @Entity mapping

```
@Entity
public class SavingsAccount extends Account {
   private Double APY;

   ...
```

Java.util.Double instead of primitive double type








```
@Entity
public class CheckingAccount extends Account {
   private Double overdraftLimit;

   ...
```

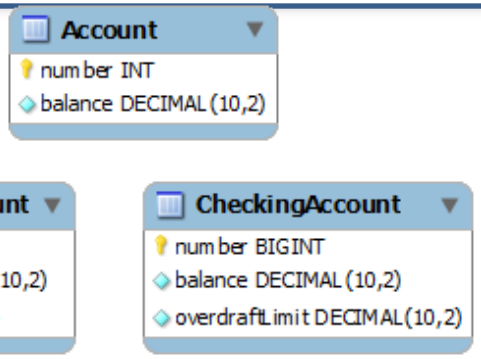
Java.util.Double instead of primitive double type

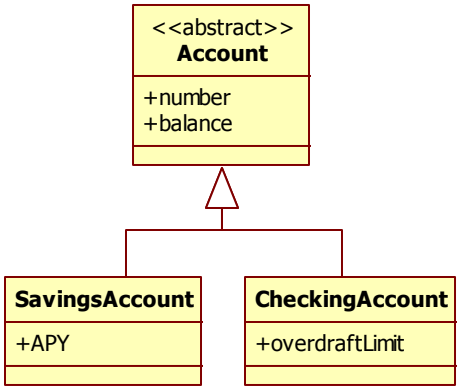
XML

```
<hibernate-mapping package="concrete">
  <class name="Account" abstract="true">
    <id name="number">
      <generator class="hilo" />
    </id>

    <property name="balance" />

    <union-subclass name="SavingsAccount">
      <property name="APY" />
    </union-subclass>

    <union-subclass name="CheckingAccount">
      <property name="overdraftLimit" />
    </union-subclass>
  </class>
</hibernate-mapping>
```

Identity generation can not use identity column

<union-subclass> tag





Inheritance Mapping

WRAPPING UP

Recommendations

- If subclasses don’t contain a lot of properties use single table per hierarchy
- If subclasses have many properties it is generally best to use joined tables
- Generally avoid using table per concrete class unless you do not have any polymorphic associations

Active Learning

- What are the advantages and disadvantages of the joined table strategy?

- Why should we not specify @Id on sub classes?

Module Summary

- In this module we covered the different ways to map inheritance with Hibernate
  - Single Table: De-normalized schema, but very efficient queries
  - Joined Tables: Normalized schema, but less efficient queries
  - Table per Concrete: has some issues, but is very efficient if polymorphism is not a priority
- Which strategy to use depends on your business needs, when in doubt the joined table is the most flexible, although slower