Using the interfaces below, write a legal bean class. You don't have to write the actual business logic, but at least list all the methods that you have to write in the class, with their correct declarations.

<<interface>> ***ProductHome***
create(String description, String cat, double price, String ID)
findByPrimaryKey(String key)
findByCategory(String category)
getLowStockItems()

<<interface>> ***Product***
getCategory()
getID()
getDescription()
setDescription()
getPrice()
setPrice()

<<interface>> ***EntityBean***
setEntityContext(EntityContext ec)
ejbActivate()
ejbPassivate()
ejbRemove()
unsetEntityContext()
ejbLoad()
ejbStore()

```
public abstract class CustomerBeanCMP implements EntityBean  {

  public String ejbCreate(String cat, double price, String ID) {
    return null;
  }

  public abstract String getProductCategory();
  public abstract void setProductCategory(String cat);
  public abstract String getProductID();
  public abstract void setProductID(String ID);
  public abstract String getProductDescription();
  public abstract void setProductDescription(String desc);
  public abstract double getProductPrice();
  public abstract void setProductPrice(int price);

  public String getCategory() {
    return this.getProductCategory();
  }

  public void setCategory(String cat) {
    this.setProductCategory(cat);
  }

  public String getID() {
    return this.getProductID();
  }

  public void setID(String ID) {
    this.setProductID(ID);
  }

  public String getDescription() {
    return this.getProductDescription();
  }

  public void setDescription(String desc) {
    this.setProductDescription(desc);
  }

  public double getPrice() {
    return this.getProductPrice();
  }

  public void setPrice(double price) {
     this.setProductPrice(price);
  }

  public void ejbPassivate() { }
  public void ejbActivate() { }
  public void ejbRemove() { }
  public void unsetEntityContext() { }
  public void ejbLoad() { }
  public void ejbStore() { }
  public void setEntityContext(EntityContext ctx) { }

  public void ejbPostCreate(String cat, double price, String ID) { }

  public Collection ejbHomeGetLowStockItems() {
      // return a Collection
  }
}
```

NOTE: we don't put the Finder methods in a CMP class!

NOTE: the abstract methods have a different name than the publicly-exposed methods in the interface. You *can* expose the abstract CMP field methods directly to the client, but remember -- that's a really bad idea. So we didn't.

Complete code for the CustomerBeanCMP class

(note: we aren't showing the annotation, even here in the answers. That's still YOU job.)

```
package headfirst;

import javax.ejb.*;

public abstract class CustomerBeanCMP implements EntityBean {

   private EntityContext context;

H  public String ejbCreate(String last, String first, String addr) {
     this.setLast(last);
     this.setFirst(first);
     this.setPK(makePK());
     this.setAddress(addr);
     return null;
   }

C  public String getLastName() {
      return this.getLast();
   }

C  public void setLastName(String name) {
      this.setLast(name);
   }

C  public String getFirstName() {
      return this.getFirst();
   }

C  public void setFirstName(String name) {
      this.setFirst(name);
   }

C  public String getAddress() {
      return this.getCustAddress();
   }

C  public void setAddress(String addr) {
      this.setCustAddress(addr);
   }

EB✔public void setEntityContext(EntityContext ctx) {
      context = ctx;
   }
```

```
VF  public abstract String getLast();
    public abstract void setLast(String last);
    public abstract String getFirst();
    public abstract void setFirst(String first);
    public abstract String getCustAddress();
    public abstract void setCustAddress(String addr);
    public abstract String getPK();
    public abstract void setPK(String pk);

✔ EB public void unsetEntityContext() { }
    public void ejbLoad() { }
    public void ejbStore() { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void ejbRemove() { }

?   private String makePK() {
      int rand = (int) (Math.random() * 42);
      return ""+ rand;
    }
  }
```

this is just our own private method, so it doesn't come from any rule other than our business logic

1. Mark each method in the CustomerBeanCMP class with one of the following four symbols:

   H C EB VF

   based on the reason for that method's existence in the class. For example, the ejbCreate() method is required because there's a matching create() in the home, so mark an H next to the ejbCreate() method.

2. Put a check mark next to those methods that the compiler cares about. In other words, if you left a method out and the compiler would complain with an error, then mark that method with a ✔

3. Annotate the code yourself with any other details you can think of. For this exercise (but not the previous two), do as much as you can on your own, then turn back to earlier pages in this chapter and see if you can add or change anything.

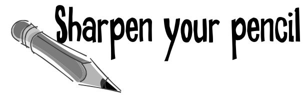
there are no Dumb Questions

Q: It looks like there are TWO ways to move to the method-ready state: either the client calls a create() method or the Container calls ejbActivate(). So does this mean that you can't count on ejbActivate() being called each time you leave the pool?

A: That's right. A bean can move to the method-ready state by ONLY those two paths (creation or activation) but never both at the same time. So, if you have a design that acquires resources in ejbActivate(), so that they'll always be available while the bean is servicing a business method, you better grab them in ejbCreate() (or ejbPostCreate()—you'll see the difference in a few minutes).

In the real world, it's much less common in EJB 2.0 to use ejbActivate() for much of anything. We'll talk more about this both in this chapter and the last chapter (patterns and performance), but the short version is this: it's usually more efficient to acquire and release scarce resources just within the business methods that need them. That way, you're not hanging on to them (preventing other beans from having access) while your bean is active (i.e. not in the pool), but not actively running a method. Yes, that means you have some additional overhead in each business method, as opposed to grabbing the thing once in ejbActivate(), but in many cases the overhead of grabbing the resource is minor compared to the scalability cost of holding resources (we're thinking... database connections from the pool) open longer than you need to access those resources.

Bottom line: You'll probably find yourself leaving ejbActivate() empty, in so you won't have to worry about missing it when you come out of the pool via an ejbCreate() call.



For the exam, you have to know exactly which container callback methods are in the EntityBean interface, so you need to memorize these. The tricky part is that some of them have the same names, but completely different behavior than their session bean counterparts in the SessionBean interface. DO NOT LOOK ON THE OPPOSITE PAGE!

1. The client calls this method to tell the Container that he (the client) is done using the bean's EJB object reference:

Ah-hah! There isn't one! (trick question)

2. This method is called when the bean goes back to the pool, after an entity is deleted from the database:

Another trick... ejbRemove() is called BEFORE the entity is deleted, but ejbPassivate() is not called after that.

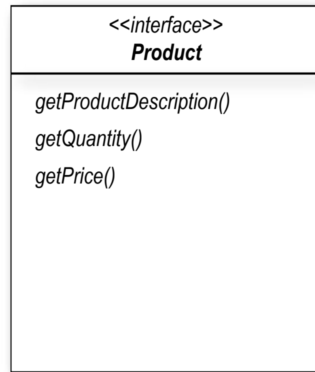
3. This method is called immediately after the bean's constructor runs:

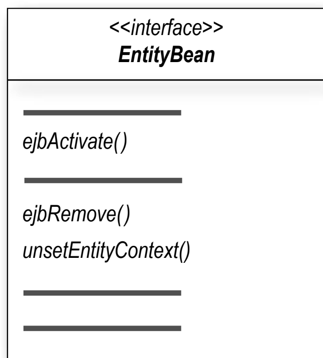
setEntityContext()

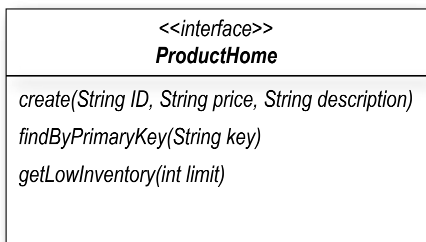
4. This method is called on the bean when the bean is in the pool, and the client invokes a business method on the component interface. (We're looking for the *first* method called in that scenario)

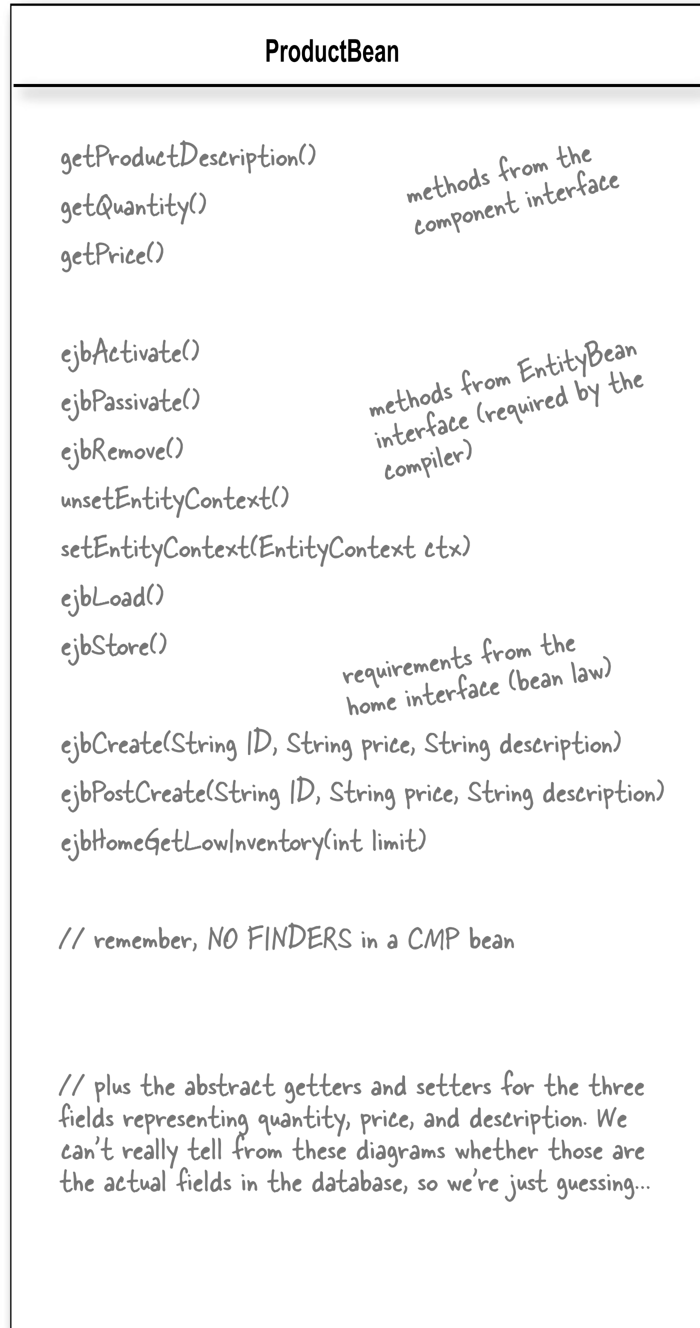
ejbActivate() (followed by ejbLoad())

Fill in the ProductBean UML-ish box with the methods that YOU must write in your bean class, given the component and home interfaces. Don't forget the container callbacks from EntityBean, although we've shown you only three of the seven. The rest you'll have to remember and fill in.



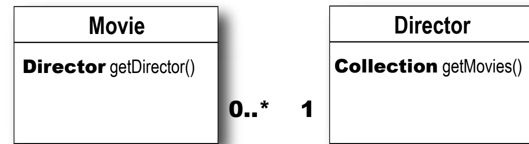






Sharpen your pencil

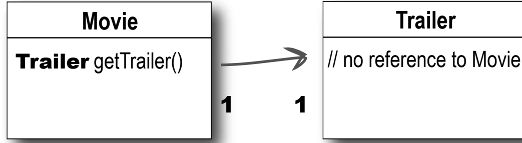
Here's the relationship DD for Director-to-Movie, but we've left a few things out. See if you can fill them in correctly *without* looking on the previous pages!



Director-to-Movie relationship

```
<relationships>
   <ejb-relation>
     <ejb-relationship-role>
       <ejb-relationship-role-name>  DirectorBean  </ejb-relationship-role-name>
       <multiplicity>  One  </multiplicity>
       <relationship-role-source>
         <ejb-name>DirectorBean</ejb-name>
       </relationship-role-source>
       <cmr-field>
         <cmr-field-name>  movies  </cmr-field-name>
         <cmr-field-type>java.util.Collection</cmr-field-type>
       </cmr-field>
     </ejb-relationship-role>
     <ejb-relationship-role>
       <ejb-relationship-role-name>MovieBean</ejb-relationship-role-name>
       <multiplicity>  Many  </multiplicity>
       <cascade-delete />
       <relationship-role-source>
         <ejb-name>  MovieBean  </ejb-name>
       </relationship-role-source>
       <cmr-field>
         <cmr-field-name>director</cmr-field-name>
       </cmr-field>
     </ejb-relationship-role>
   </ejb-relation>
</relationships>
```

we have to have this because the multiplicity of the OTHER partner is Many, which means the cmr field is a collection, and we have to say what *type* of collection (Set or Collection)



Relationships can be one-way (unidirectional)

You can have a relationship between two beans, but have a CMR field in only one of the two beans. For example, if you set up a relationship between a Movie and its Trailer (a one-to-one relationship), and you don't want clients to use a Trailer to get to a Movie, just leave the CMR field for Movie out of the Trailer bean. Simple as that.

In that case, the TrailerBean won't know anything about the MovieBean, even though they're both partners in a relationship.

```
<ejb-relationship-role>
    <ejb-relationship-role-name>TrailerBean</ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <cascade-delete />
    <relationship-role-source>
      <ejb-name>TrailerBean</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name></cmr-field-name>
    </cmr-field>
 </ejb-relationship-role>
```

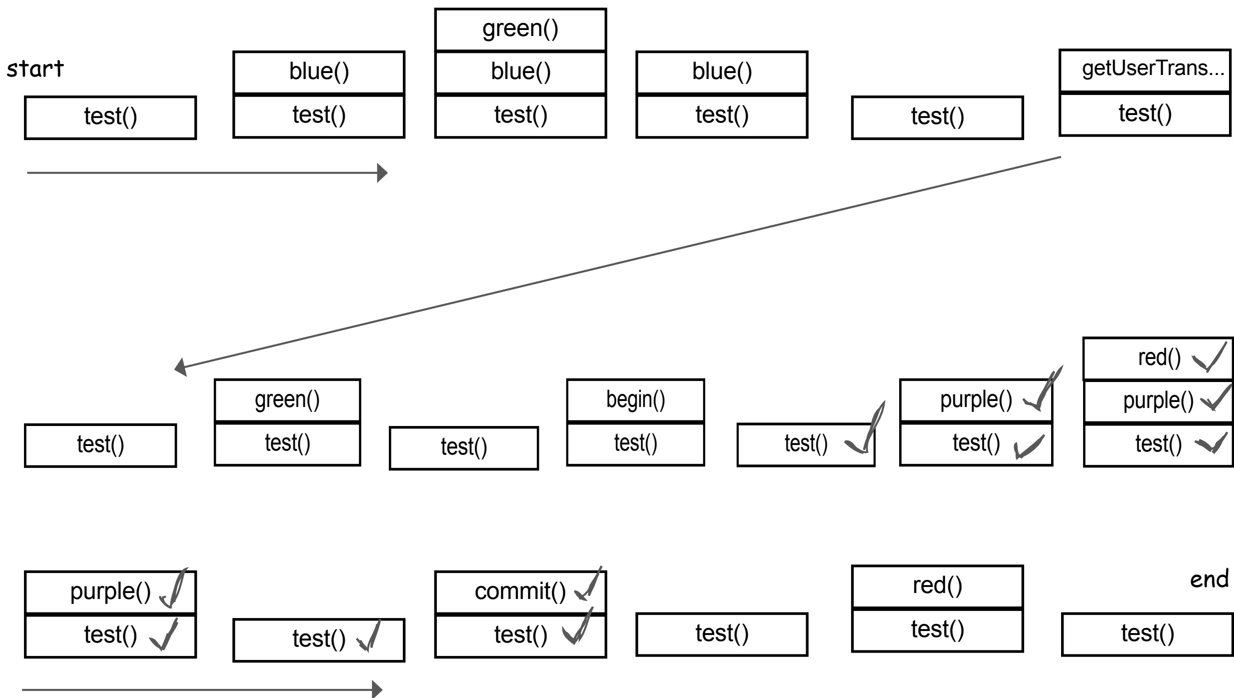
Leave the cmr-field out of a relationship role if you don't want this bean to have a reference to its partner bean.

Sharpen your pencil

Using this code listing, mark the matching call stack frames with a checkmark if that frame is currently in a transaction. We did one in the middle for you.

```
public void test() throws Exception {
   blue();
   UserTransaction ut = ctx.getUserTransaction();
   green();
   ut.begin();
   purple();
   ut.commit();
   red();
}

void blue() { green(); }
void green() { }
void purple() { red(); }
void red() { }
```



For the exam (and bean developer life in general) you have to know some REALLY important rules about transactions, and it will be much easier for you if you take the time now to figure some of this out for yourself. Understanding is much better than memorizing, and it's not like you don't have enough to memorize as it is. You'll find all of these questions answered over the next few pages, but you should *really* try to do this first.

1. **Of the six transaction attributes, which one (or ones) must NOT be used by a bean that calls getRollbackOnly() or setRollbackOnly()?**

   Never

2. **Which transaction attribute (or attributes) must NOT be used by a message-driven bean?**

   (Hint: remember, a message-driven bean doesn't have a "client"; the container invokes the onMessage() method.)

   Mandatory, Supports, RequiresNew, Never

3. **Under what circumstances do you think the container should automatically roll back a transaction?**

   If the bean gets a runtime exception?

   If the bean throws an application exception? (e.g. InsufficientFundsException?)

   runtime exception... it's unexpected!

4. **Of the six transaction attributes, three of them can be dangerous, with one in particular being EXTREMELY risky. Keeping in mind that the Bean Provider is NOT the one who specifies the attributes for the bean's methods, which of the six is potentially the most dangerous?**

   Mandatory and NotSupported are dangerous, because they throw exceptions, but Supports is the riskiest.... because it means at runtime you don't know whether it will run in a transaction or not.

answer to the Attributes sharpen: three and five