We’ve refactored `statement()` and added `htmlStatement()` with minimal duplication.

After class, I applied the Extract Method/Move Method refactoring pair to the calculation of frequent renter points, which is now done in the `Rental` class.
Movie is still a pure "data" classes -- no behavior.
Rental contributes useful behavior.
Customer uses Rentals to do its work.
Now, the client is ready to change her pricing strategy...

*create new categories*

*change existing categories*

*others?*

The next change is here, or almost. Is the code ready?
... but this code isn’t ready:

```java
public class Rental {
    ...
    public double charge() {
        double total = 0;
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                total += 2;
                if (getDaysRented() > 2)
                    total += (getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                total += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                total += 1.5;
                if (getDaysRented() > 3)
                    total += (getDaysRented() - 3) * 1.5;
                break;
        }
        return total;
    }
    ...
}
```

How can we make it ready?

Modify the design so that we can add new categories of movie and change pricing strategies with ease.

Or at least more easily!

Note that we have another violation of the Law of Demeter, as well as a switch based on type of movie.
Step 1

`copy charge() to Movie`

This choice involves sending `daysRented` as an argument. The code was using one piece of data from each class. Why is it better?

- One class reveals data to another.
  A change to that implementation decision is contained within the class.
  (as opposed to using another object’s data)

- This creates new types.
  Types are more volatile, more likely to change
  
  A change to that implementation decision is contained within the class.

- A new type of object can provide other services.
  Movie contributes to the solution now!
Step 2

*change* Rental.charge() to *send a message to the Movie*

Now, we have several kinds of movie answering different ways based on their type, implemented with a flag variable and a switch statement.

This sounds like a job for subclasses!
But that won’t work. Why?

A movie can change its type during execution. A new release will become a regular release, or a regular children’s movie.

An object cannot change its class during its lifetime.

Do you remember this problem from before?
   (Session 17)
Do you remember its solution?
   (composition on the Role Object pattern ... here, the State pattern)
A Movie's price can change but substituting a different Price object into its instance variable.

A Movie's can calculate its price by sending a message to its Price object.

We can create new kinds of Price strategy by adding new subclasses.

We can change an existing Price strategy by changing a single class.
Step 3
Replace Type Code with State

*change Movie.priceCode from an int to a state object*

1. Change the constructor to use the setter method for price code.

2. Create new Price classes.
   (copy these in -- don’t type)

3. Change Movie accessors to use the Price classes.
   (copy these in -- don’t type)
   Notice how Movie **encapsulates** its use of Price classes!
Step 4
Move Method

move Movie.charge
to the Price class

Again, copy, compile/test, call, compile/test.
Step 5
Replace Conditional with Polymorphism

move one arm of Price.charge into the subclasses at a time

Again, copy the method.
Strip all but one arm of the switch.
Compile/test.
Remove old arm.
Compile/test.

At end, leave charge() as an abstract method in Price. Vars typed to Price need to respond!
new class diagram

Better! Movie works.
new class diagram

Better! Price is abstract, and instances of its subclasses do work.
new interaction diagram

Much better!!!!!
refactoring in traditional dev

versus

refactoring in agile dev

Do the simplest thing you possibly can.

Refactor to improve the code.


The design evolves slowly as we add features.
Refactoring is where we redesign!!!

[ ... fill in the blanks. Do the simplest thing. YAGNI. Refactor to create design. ]