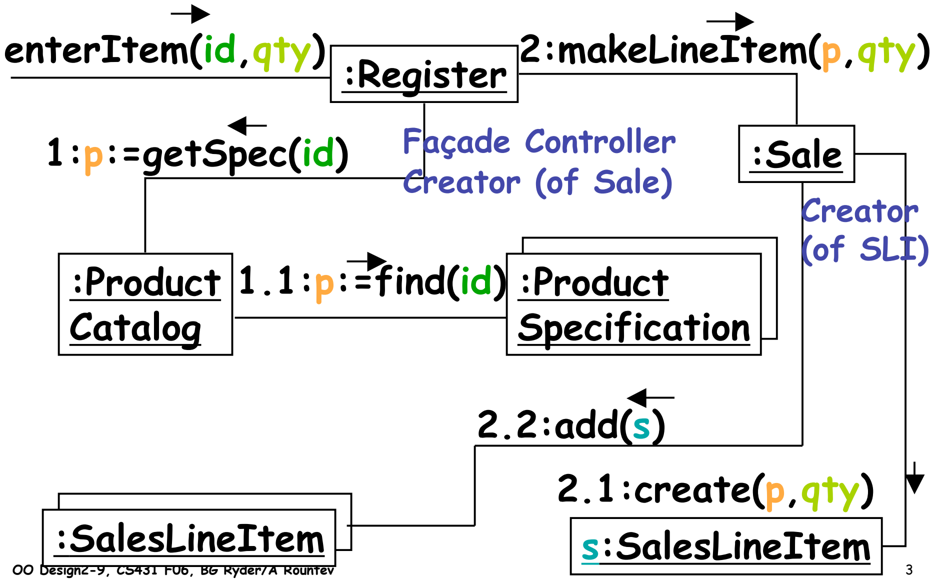
OO Design2

- POS example - revisited
    - LAR Ch 18 has entire POS design explained
    - READ THIS CHAPTER and ASK Q's in class
- Design class diagrams
    - Kinds of visibility of objects to one another
    - Navigability of associations
- How to do implementation from design artifacts?

Design Artifacts

- Design Class Diagrams (DCDs)
    - Differences from Conceptual Class Diagrams in Domain model
        - Contain types, directed associations with multiplicities, numbered actions
        - How visibility between objects is provided
- Interaction Diagrams
    - Sequence Diagrams
        - Vertical sequence format
    - Communication Diagrams
        - Network format

Communication Diagram For POS



enterItem(id,qty)

:Register

2:makeLineItem(p,qty)

1:p:=getSpec(id)

Façade Controller
Creator (of Sale)

:Sale

Creator
(of SLI)

:Product
Catalog

1.1:p:=find(id)

:Product
Specification

2.2:add(s)

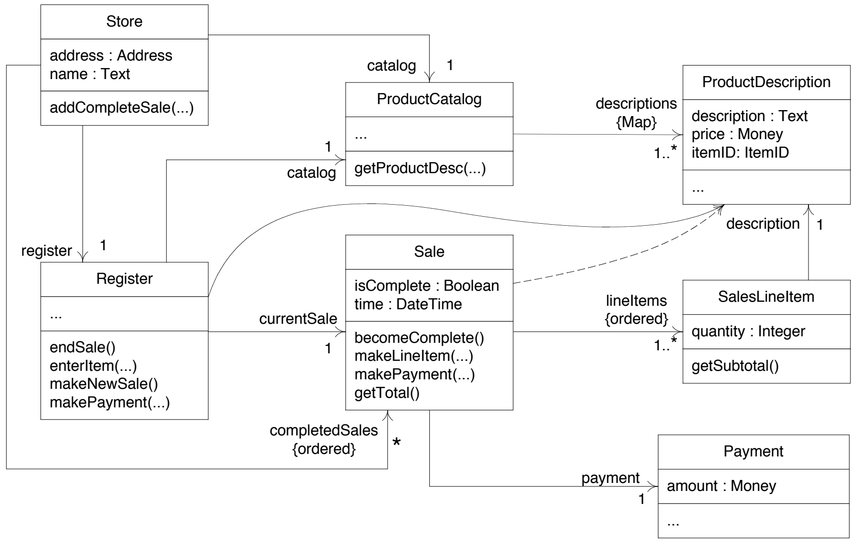
2.1:create(p,qty)

:SalesLineItem

s:SalesLineItem

Design Class Diagram for POS

(LAR, Fig 18.17)



Store
address : Address
name : Text
addCompleteSale(...)

catalog 1

ProductCatalog
...
getProductDesc(...)

descriptions
{Map}
1..*

ProductDescription
description : Text
price : Money
itemID: ItemID
...

1
catalog

register 1

Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)

description 1

Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()

currentSale
1

lineItems
{ordered}
1..*

SalesLineItem
quantity : Integer
getSubtotal()

completedSales
{ordered}
*

payment
1

Payment
amount : Money
...

Visibility between Objects

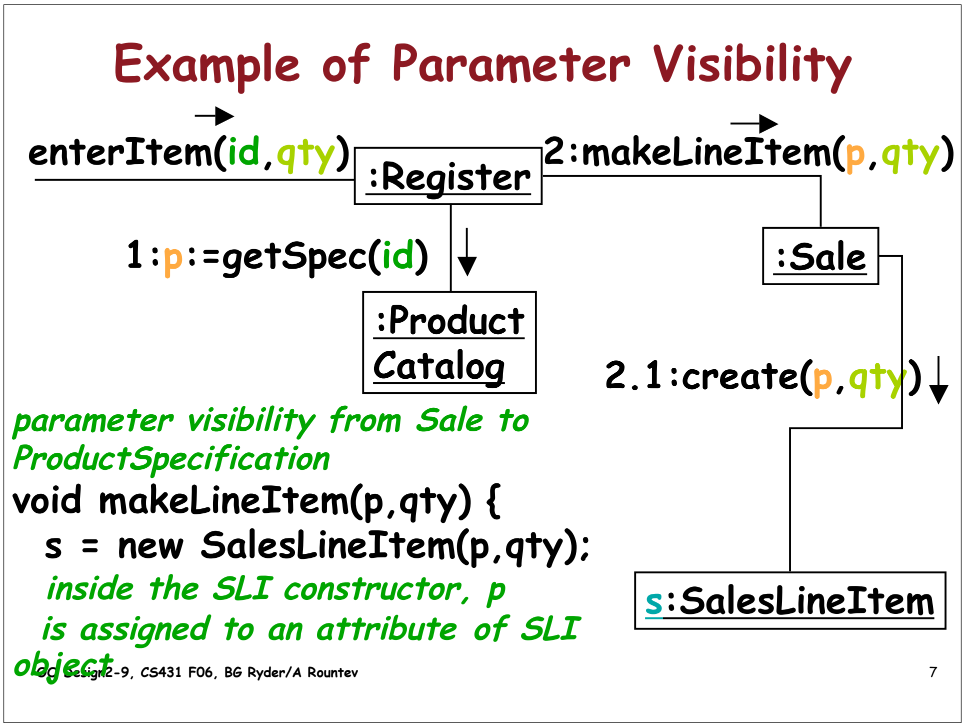
- If object A sends a message to object B, then B must be visible to A
  - i.e., A should have access to a reference (a pointer) to B
- Ensure the necessary visibility
  - If the interaction diagram shows a message, need to choose the appropriate visibility mechanism to make the message possible

Attribute & Parameter Visibility

- Reference to B is an attribute of A
  - Relatively permanent: often exists for the lifetime of the objects (common)
    - E.g., Register needs to send getSpec(id) to ProductCatalog

```
class Register {
  private ProductCatalog catalog; ... }
```

- Reference to B is a parameter to a method of A
  - Relatively temporary: exists only for the scope of the method (2nd most common)
    - Often turned into an attribute



Example of Parameter Visibility

enterItem(id,qty) :Register 2:makeLineItem(p,qty)

1:p:=getSpec(id) ↓

:Product Catalog

:Sale

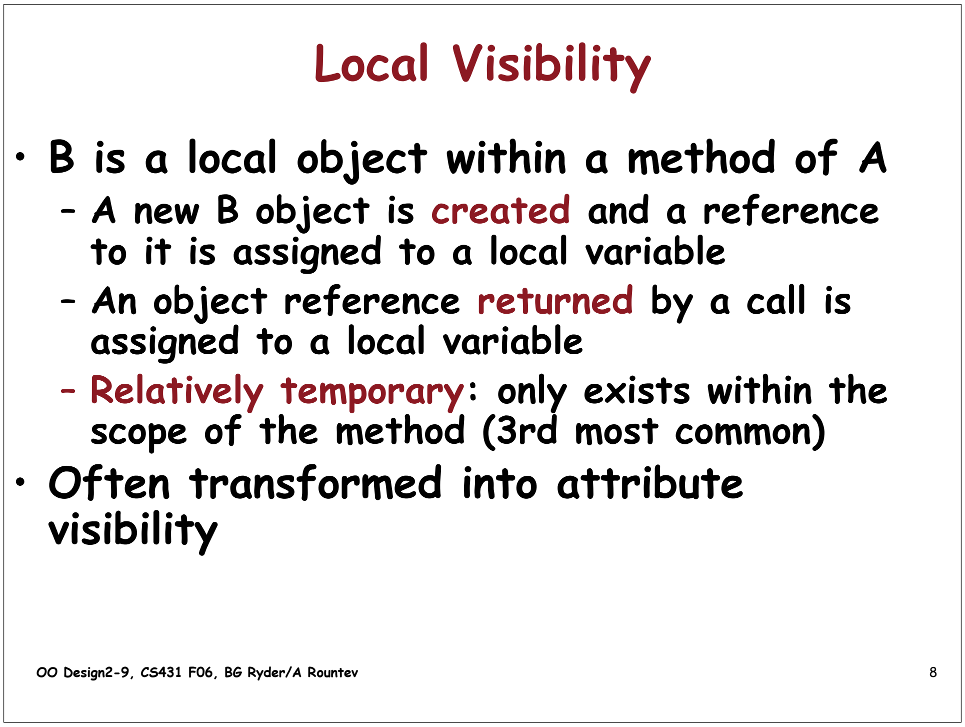
2.1:create(p,qty) ↓

s:SalesLineItem

parameter visibility from Sale to ProductSpecification

void makeLineItem(p,qty) {
  s = new SalesLineItem(p,qty);

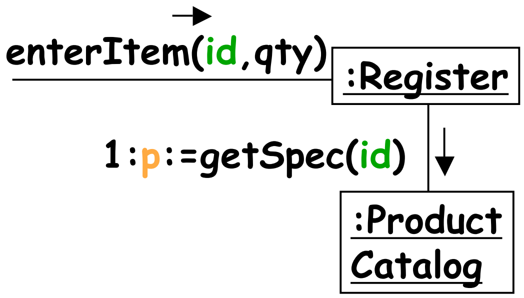
inside the SLI constructor, p is assigned to an attribute of SLI object



Local Visibility

- B is a local object within a method of A
  - A new B object is created and a reference to it is assigned to a local variable
  - An object reference returned by a call is assigned to a local variable
  - Relatively temporary: only exists within the scope of the method (3rd most common)
- Often transformed into attribute visibility

Example of Local Visibility



enterItem(id,qty)

:Register

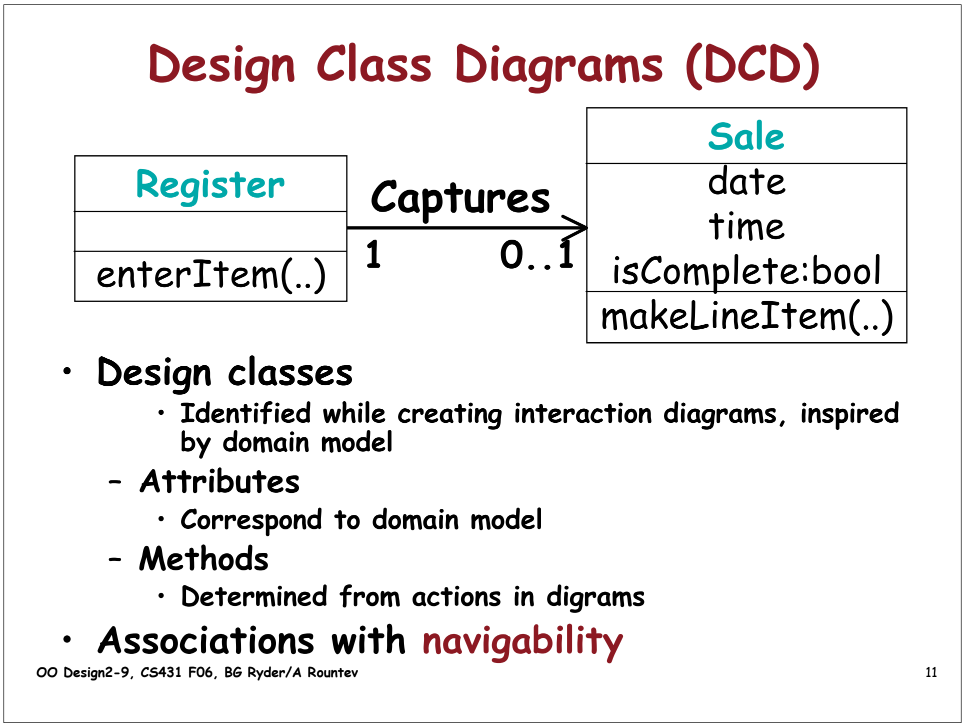
1:p:=getSpec(id)

:Product Catalog

```
enterItem(id,qty) {
   local visibility from Register
   to ProductSpecification
   p = catalog.getSpec(id); . . .
}
```

Global Visibility

- B is defined in a scope that encloses A's scope
  - E.g., a static field is "global" for all methods inside its declaring class
  - Relatively permanent: typically persists as long as A and B exist (least common)
- Should be used cautiously: may violate the principles of object orientation
- Should use Singleton pattern instead



Design Class Diagrams (DCD)

Register

enterItem(..)

Captures

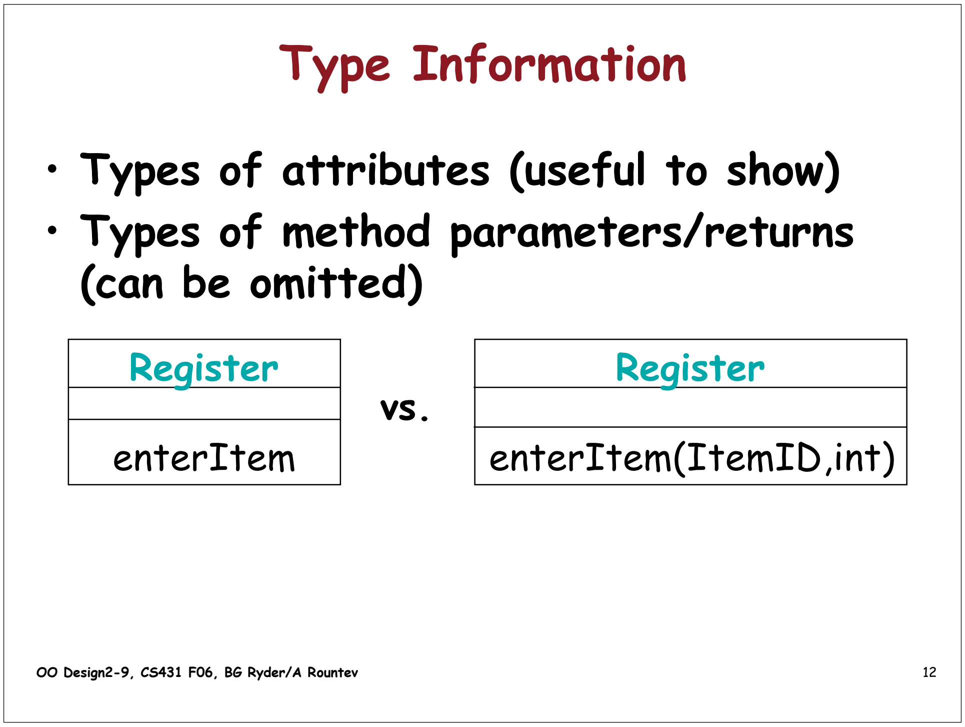
1 0..1

Sale

date
time
isComplete:bool

makeLineItem(..)

- Design classes
  - Identified while creating interaction diagrams, inspired by domain model
  - Attributes
    - Correspond to domain model
  - Methods
    - Determined from actions in digrams
- Associations with navigability



Type Information

- Types of attributes (useful to show)
- Types of method parameters/returns (can be omitted)

Register

enterItem

vs.

Register

enterItem(ItemID,int)

“create” messages

- create messages:
  - Language-independent
  - No create methods in the design classes
- For many languages: constructor(s)
  - Sometimes people do not show constructors in the DCD: reduces the clutter

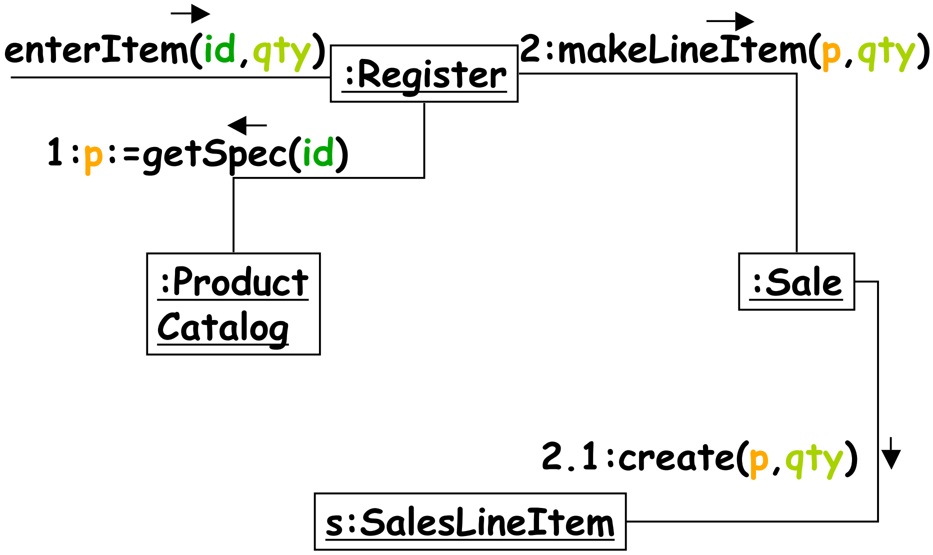
getters and *setters* for attributes

- Internal variables that implement the attribute are private and hidden
  - e.g. internally a Point attribute may be a pair of floating-point numbers
  - E.g., for price attribute of type Money
    - getPrice():Money
    - setPrice(amt:Money)
- Methods are typically not shown in design class (just show attribute)

Associations in the DCD

- Based on the interaction diagrams and the domain model
  - Often the associations already exist in the domain model
- *Will there be an ongoing, somewhat permanent connection between an instance of X and an instance of Y in order to satisfy the interactions?*
- Common cases to consider: (1) X sends a message to Y or (2) X creates Y

Partial Communication Diagram



enterItem(id,qty)

:Register

2:makeLineItem(p,qty)

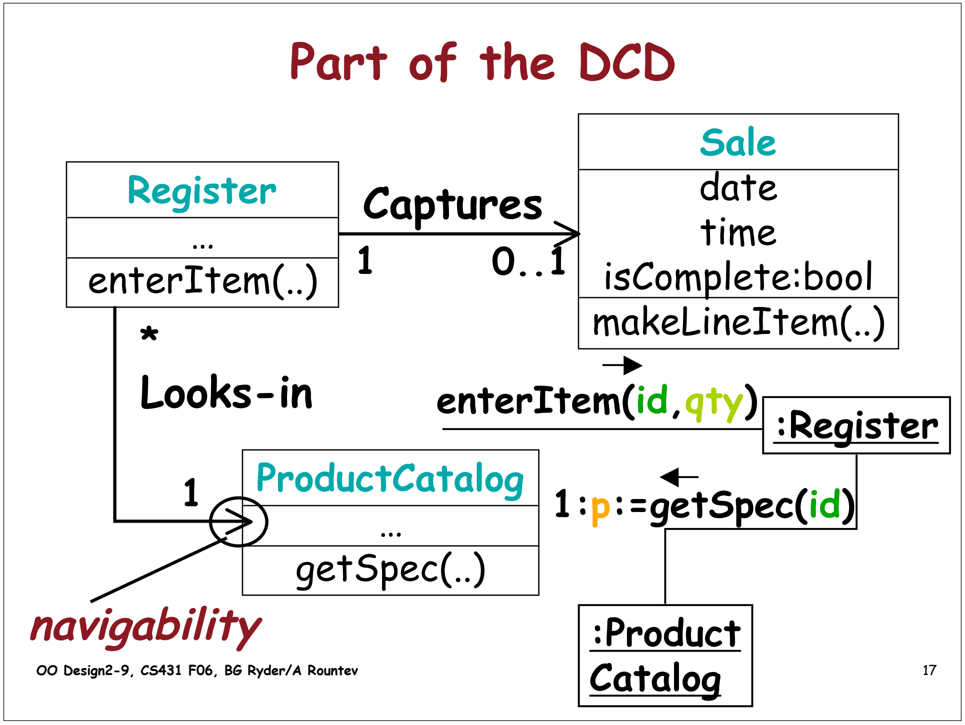
1:p:=getSpec(id)

:Product Catalog

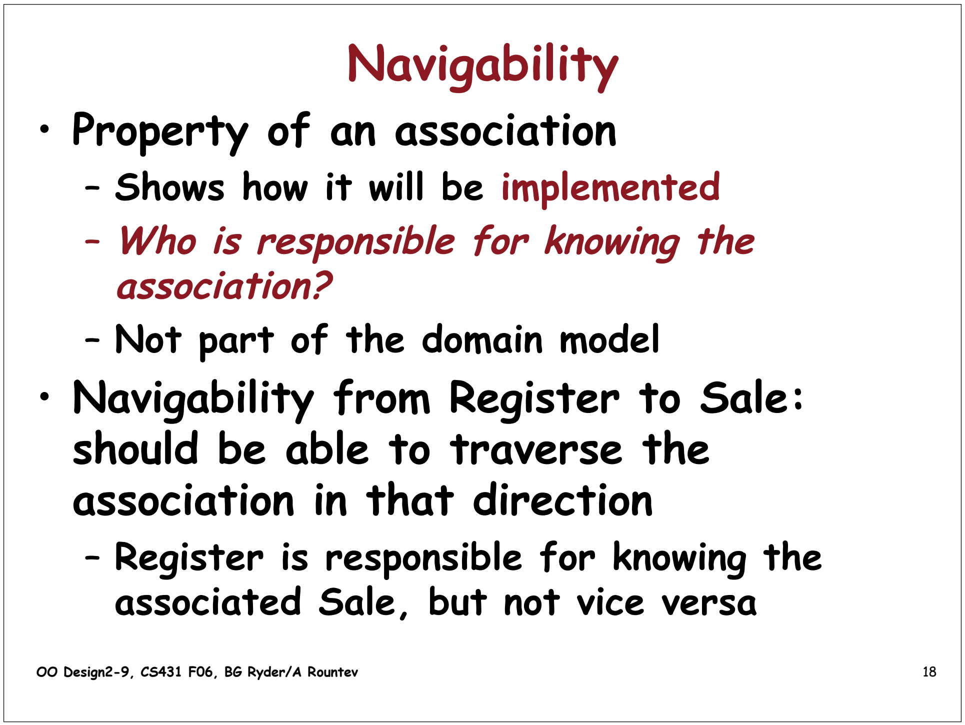
:Sale

2.1:create(p,qty)

s:SalesLineItem



Part of the DCD



Navigability

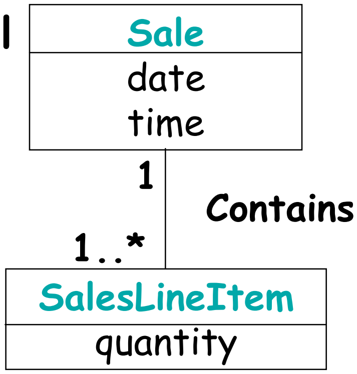
- Property of an association
  - Shows how it will be implemented
  - *Who is responsible for knowing the association?*
  - Not part of the domain model
- Navigability from Register to Sale: should be able to traverse the association in that direction
  - Register is responsible for knowing the associated Sale, but not vice versa

Navigability

- Could be 1-way or 2-way
  - X ⟷ Y
- Not mandatory, but most associations in the DCD should have it
- Implies attribute visibility
  - Will be implemented by an attribute in class Register
  - The attribute is not shown in the DCD: it is implied by the navigability

Creating a Container

- When creating this interaction diagram, we also considered the domain model

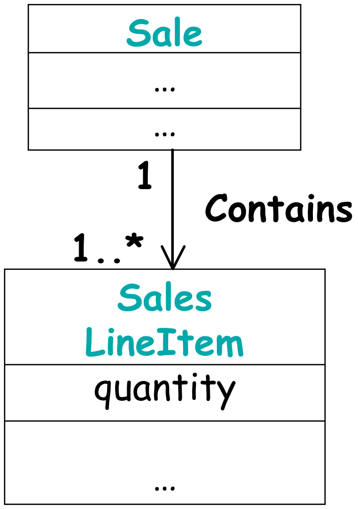


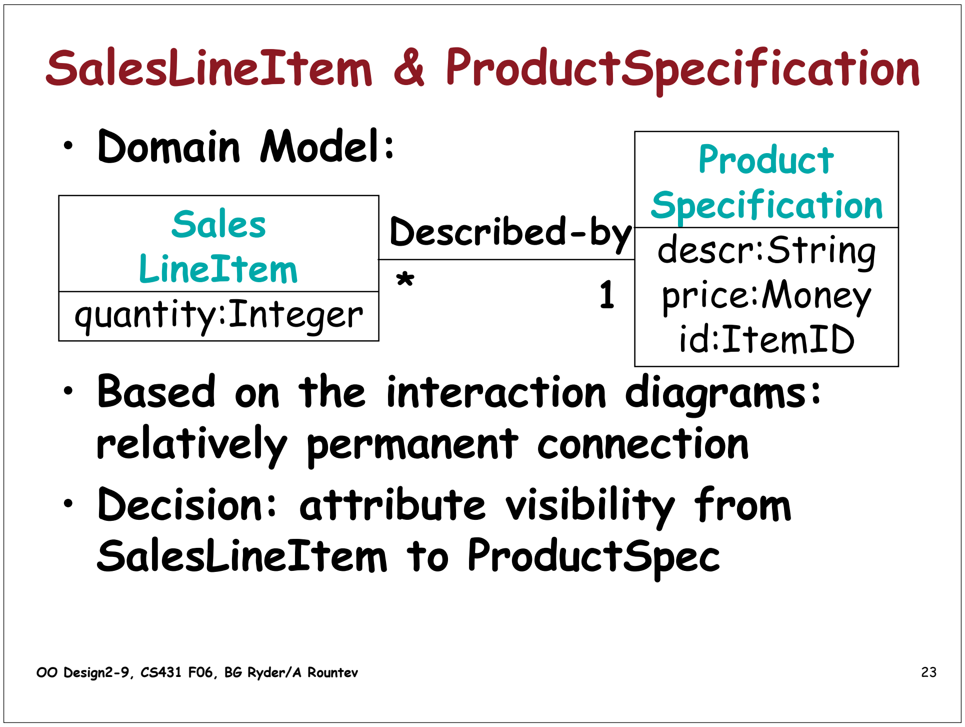
Creating a Container

- Based on the domain model: decided to use a container for SalesLineItems
  - Sale will create the container
  - This will happen when Sale is created
- Very common case for one-to-many associations: an attribute of Sale refers to the container
  - Attribute visibility from Sale to the container

Representation in the DCD

- Not necessary to show a separate container class
- The navigability implies that Sale has an attribute that refers to a set of SalesLineItem objects
  - i.e., to a container storing these objects

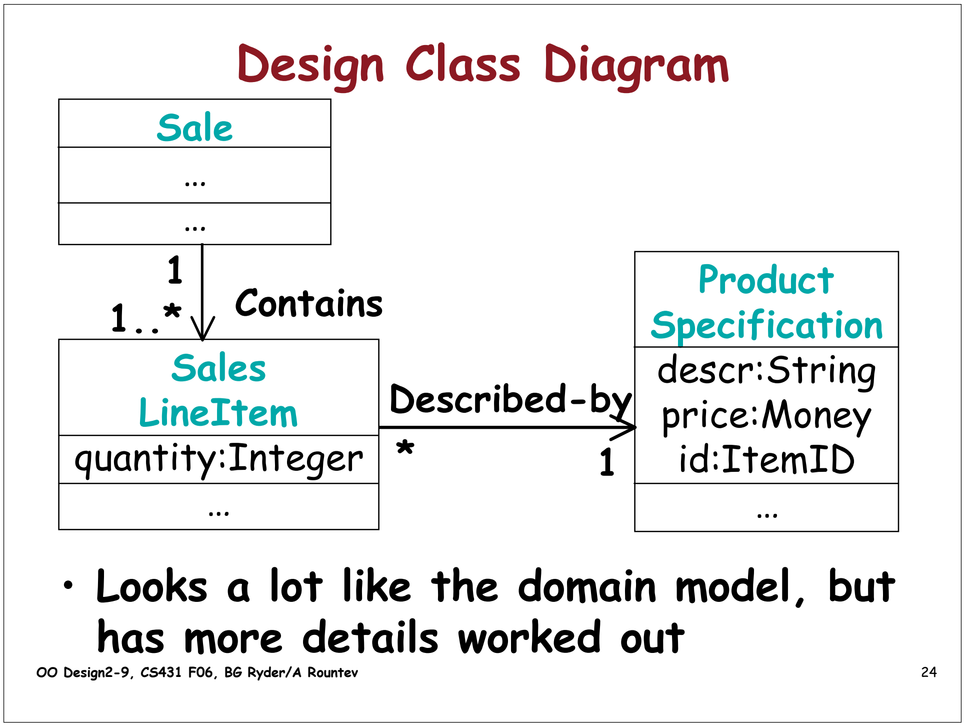




SalesLineItem & ProductSpecification

- Domain Model:

- Based on the interaction diagrams: relatively permanent connection
- Decision: attribute visibility from SalesLineItem to ProductSpec



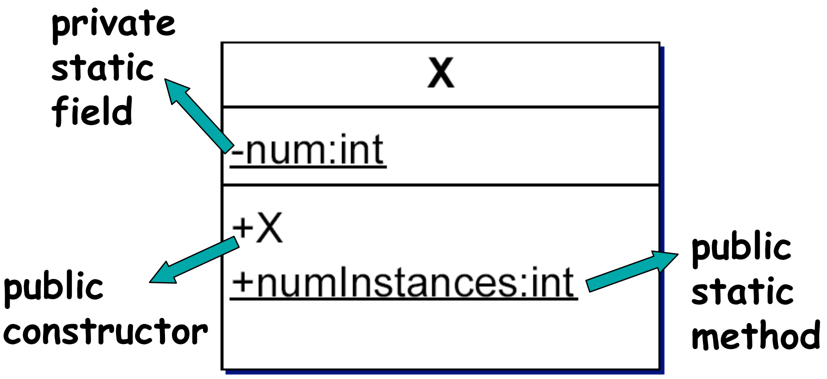
Design Class Diagram

- Looks a lot like the domain model, but has more details worked out

Accessibility of Methods and Fields

- Public: can be accessed by any code
  - UML notation: +foo
- Private: can be accessed only by code inside the class
  - UML notation: -foo
- Protected: can be accessed only by code in the class and in its subclasses
  - UML notation: #foo
- Fields usually are not public, but have getters and setters instead

UML Notation



private static field

X

-num:int

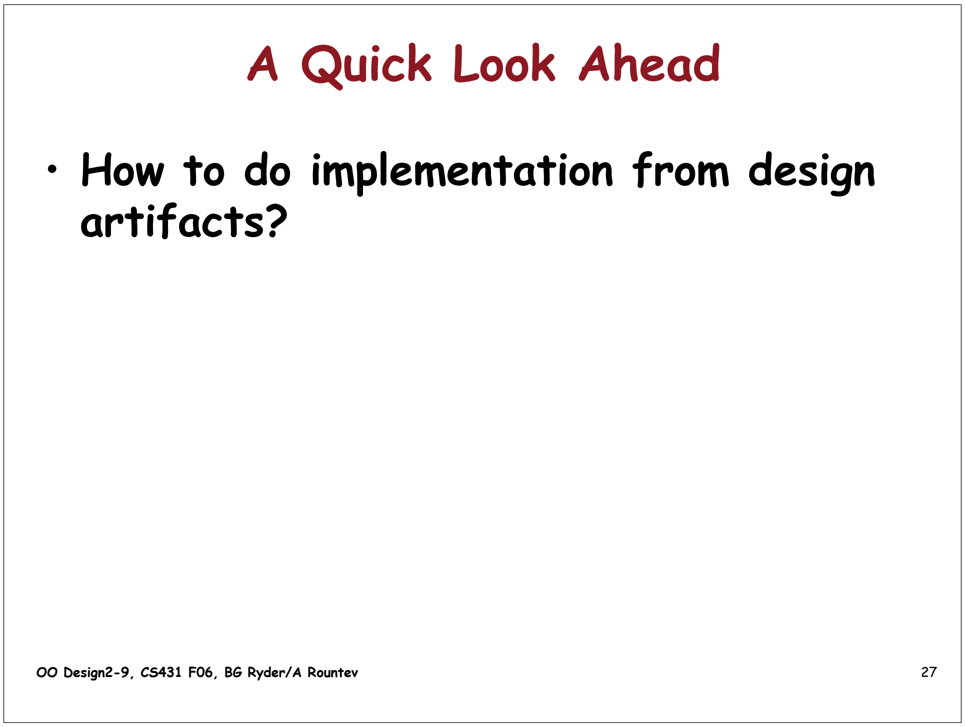
+X

+numInstances:int

public constructor

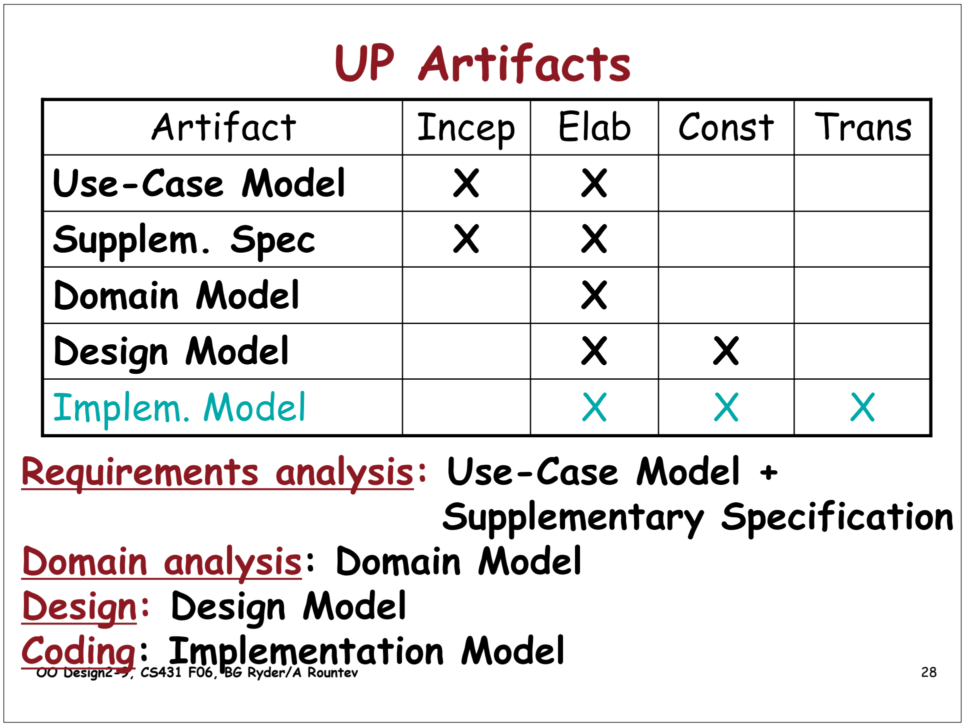
public static method

note: "static constructor"
is meaningless: by definition,
a constructor is invoked on an object



A Quick Look Ahead

- How to do implementation from design artifacts?



UP Artifacts

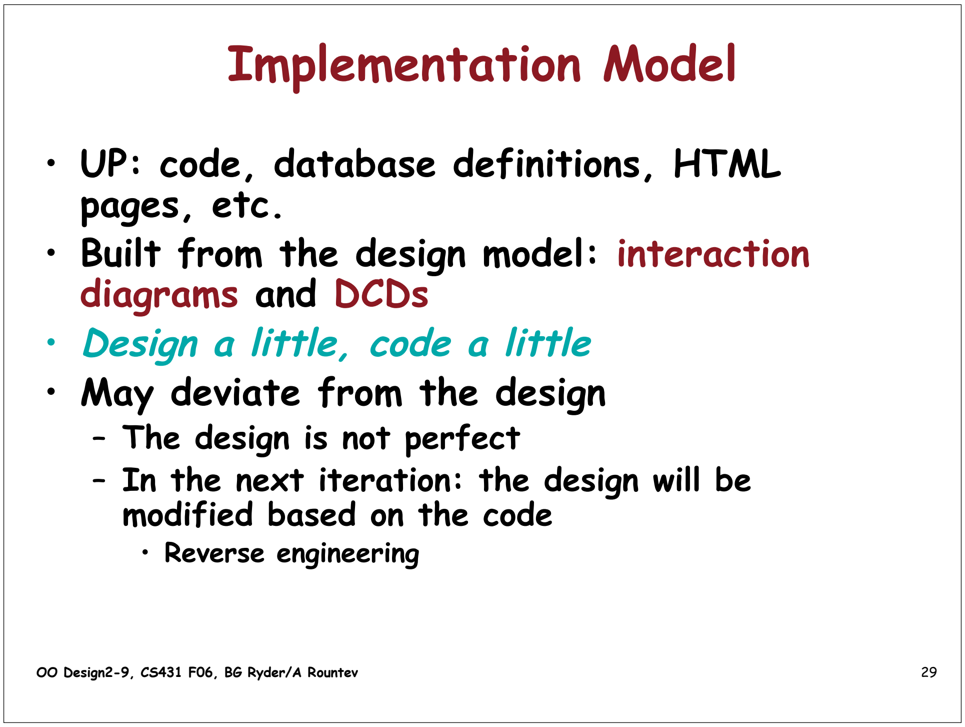
Artifact	Incep	Elab	Const	Trans
Use-Case Model	**X**	**X**		
Supplem. Spec	**X**	**X**		
Domain Model		**X**		
Design Model		**X**	**X**	
Implem. Model		X	X	X

Requirements analysis: Use-Case Model + Supplementary Specification

Domain analysis: Domain Model

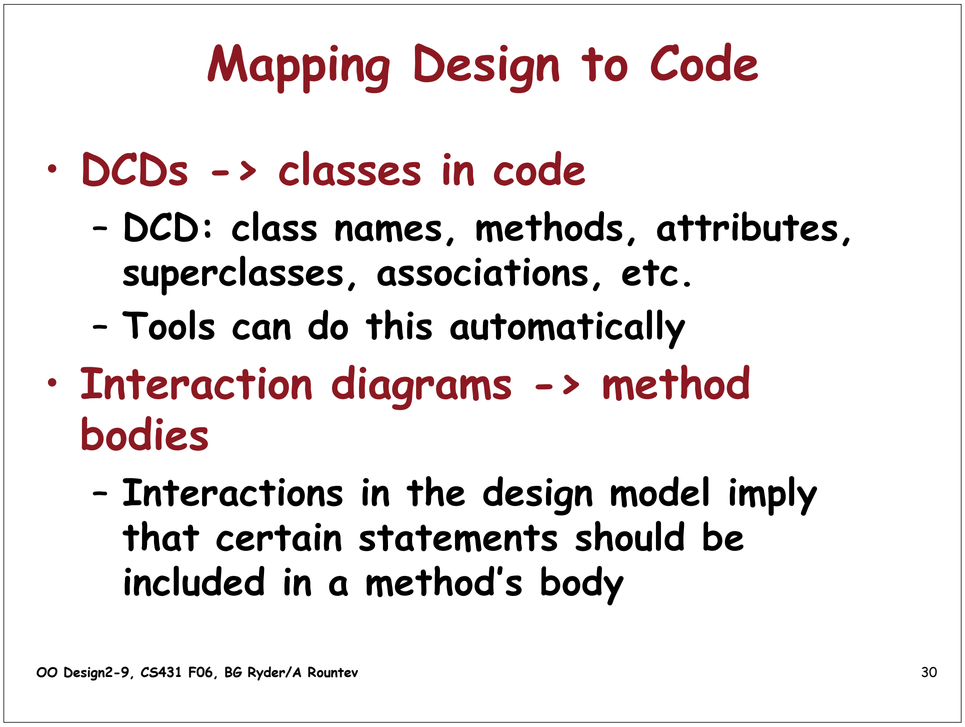
Design: Design Model

Coding: Implementation Model



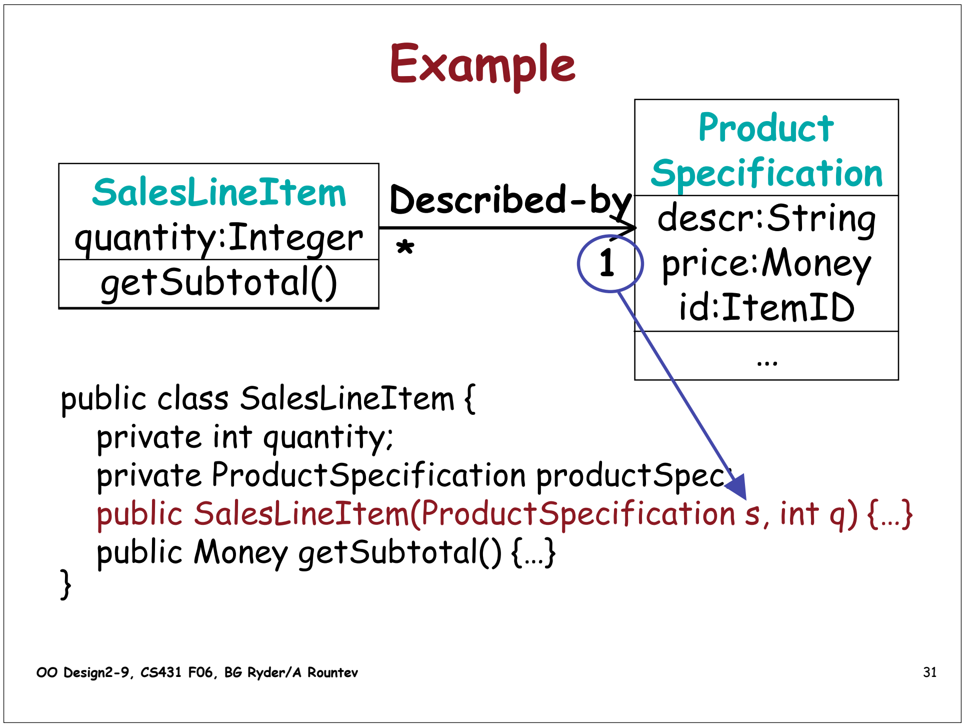
Implementation Model

- UP: code, database definitions, HTML pages, etc.
- Built from the design model: interaction diagrams and DCDs
- *Design a little, code a little*
- May deviate from the design
  - The design is not perfect
  - In the next iteration: the design will be modified based on the code
    - Reverse engineering



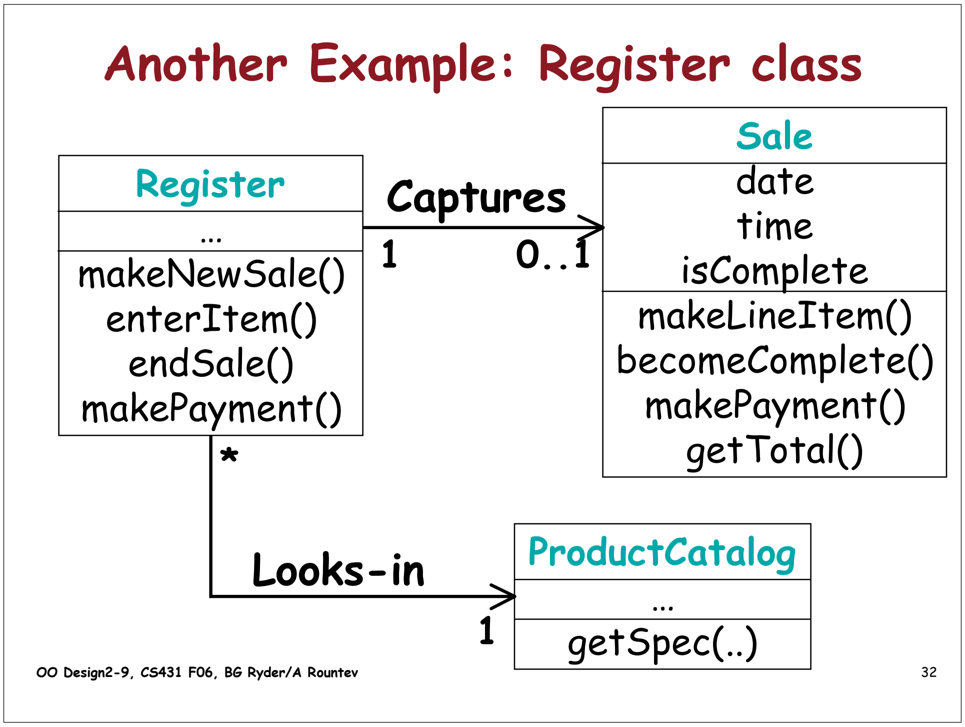
Mapping Design to Code

- DCDs -> classes in code
  - DCD: class names, methods, attributes, superclasses, associations, etc.
  - Tools can do this automatically
- Interaction diagrams -> method bodies
  - Interactions in the design model imply that certain statements should be included in a method's body

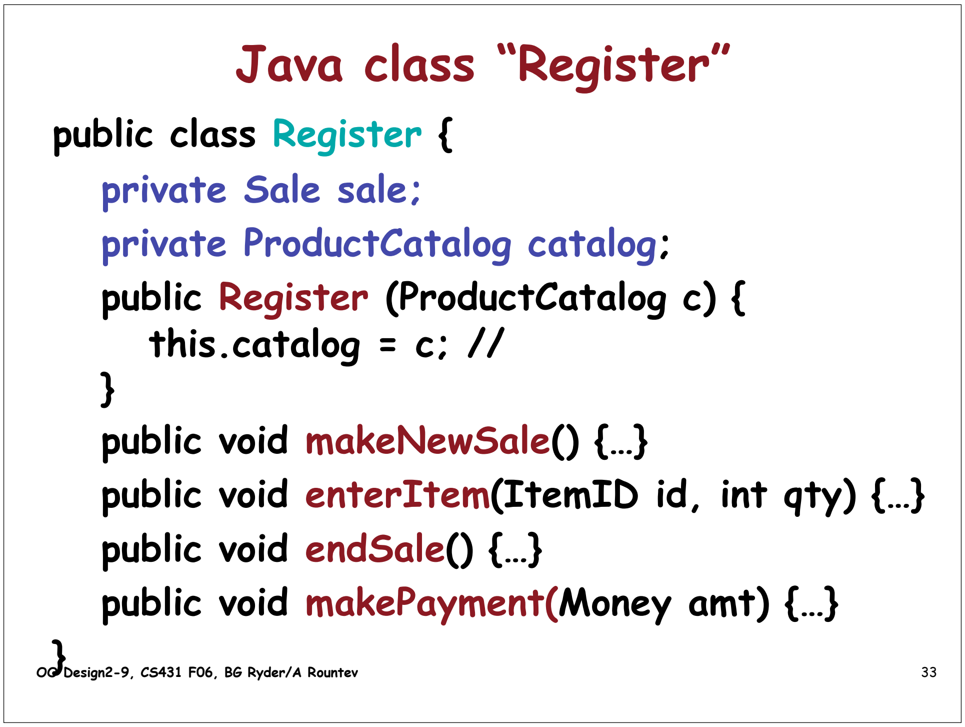


Example

```
public class SalesLineItem {
   private int quantity;
   private ProductSpecification productSpec;
   public SalesLineItem(ProductSpecification s, int q) {…}
   public Money getSubtotal() {…}
}
```

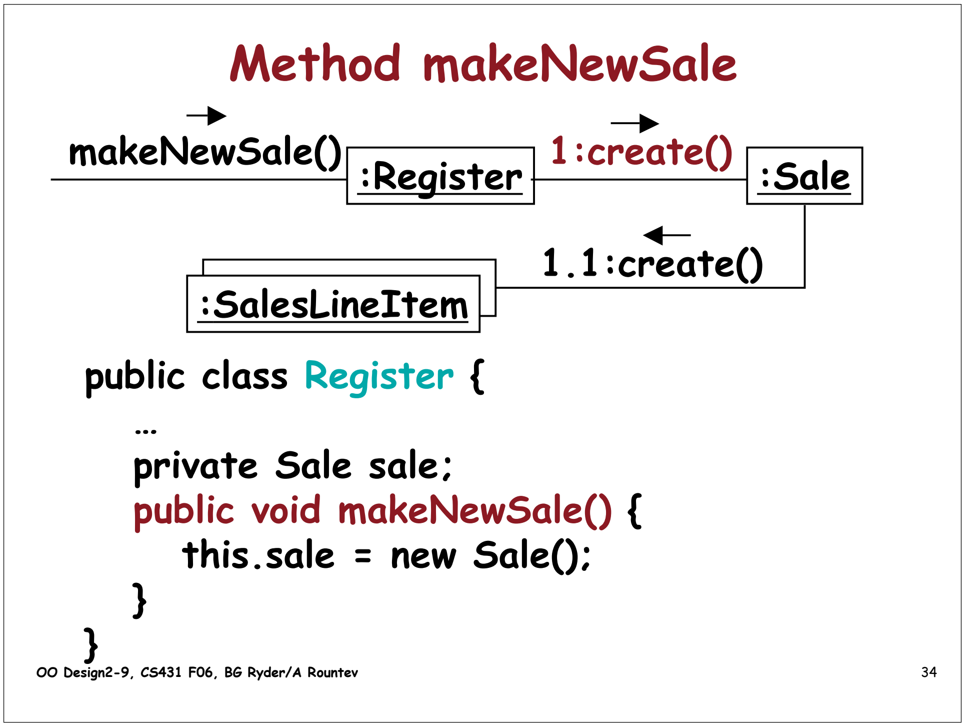


Another Example: Register class



Java class "Register"

```
public class Register {
    private Sale sale;
    private ProductCatalog catalog;
    public Register (ProductCatalog c) {
        this.catalog = c; //
    }
    public void makeNewSale() {…}
    public void enterItem(ItemID id, int qty) {…}
    public void endSale() {…}
    public void makePayment(Money amt) {…}
}
```



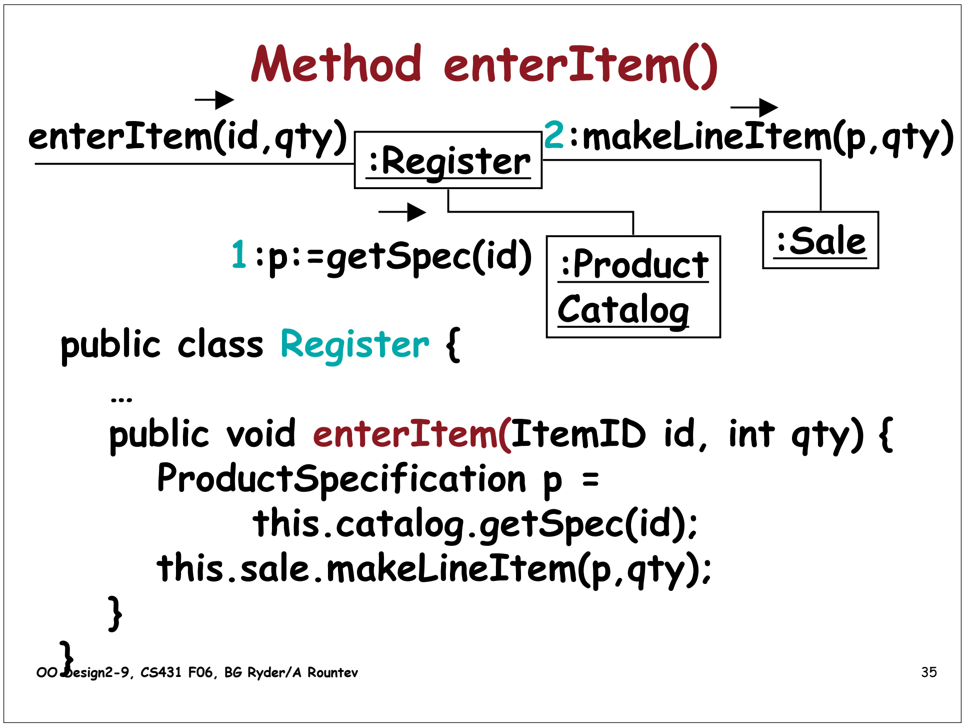
Method makeNewSale

makeNewSale() → :Register

1:create() → :Sale

1.1:create() ← :SalesLineItem

```
public class Register {
    …
    private Sale sale;
    public void makeNewSale() {
        this.sale = new Sale();
    }
}
```



Method enterItem()

enterItem(id,qty) → :Register

2:makeLineItem(p,qty) → :Sale

1:p:=getSpec(id) → :Product Catalog

```
public class Register {
    …
    public void enterItem(ItemID id, int qty) {
        ProductSpecification p =
                this.catalog.getSpec(id);
        this.sale.makeLineItem(p,qty);
    }
}
```