# MAHARISHI UNIVERSITY of MANAGEMENT

*Engaging the Managing Intelligence of Nature*

## Computer Science Department

# CS401 Modern Programming Practices (MPP)

## Professor Renuka Mohanraj

# Lecture 4: Interaction Diagrams

*Appreciating Dynamism in Silence*

# Wholeness Statement

In an OO program, objects collaborate with other objects to achieve the objectives of the program. *Sequence diagrams* document the sequence of calls among objects for a particular operation. *Object diagrams* show relationships among objects and the associations between them; they clarify the role of multiple instances of the same class. The principle of *delegation* clarifies responsibilities of each class and its instances: Requests that arrive at a particular object but cannot properly be handled by the object are *delegated* to other objects. Finally, *polymorphism* makes it possible to add new functionality without modifying existing code (as per the *Open-Closed Principle*). In these ways, we use UML diagrams to capture the dynamic features of the system; representing dynamism in the form of a static map illustrates the principle that dynamism has its basis in, and arises within, silence.

# Interaction Diagrams

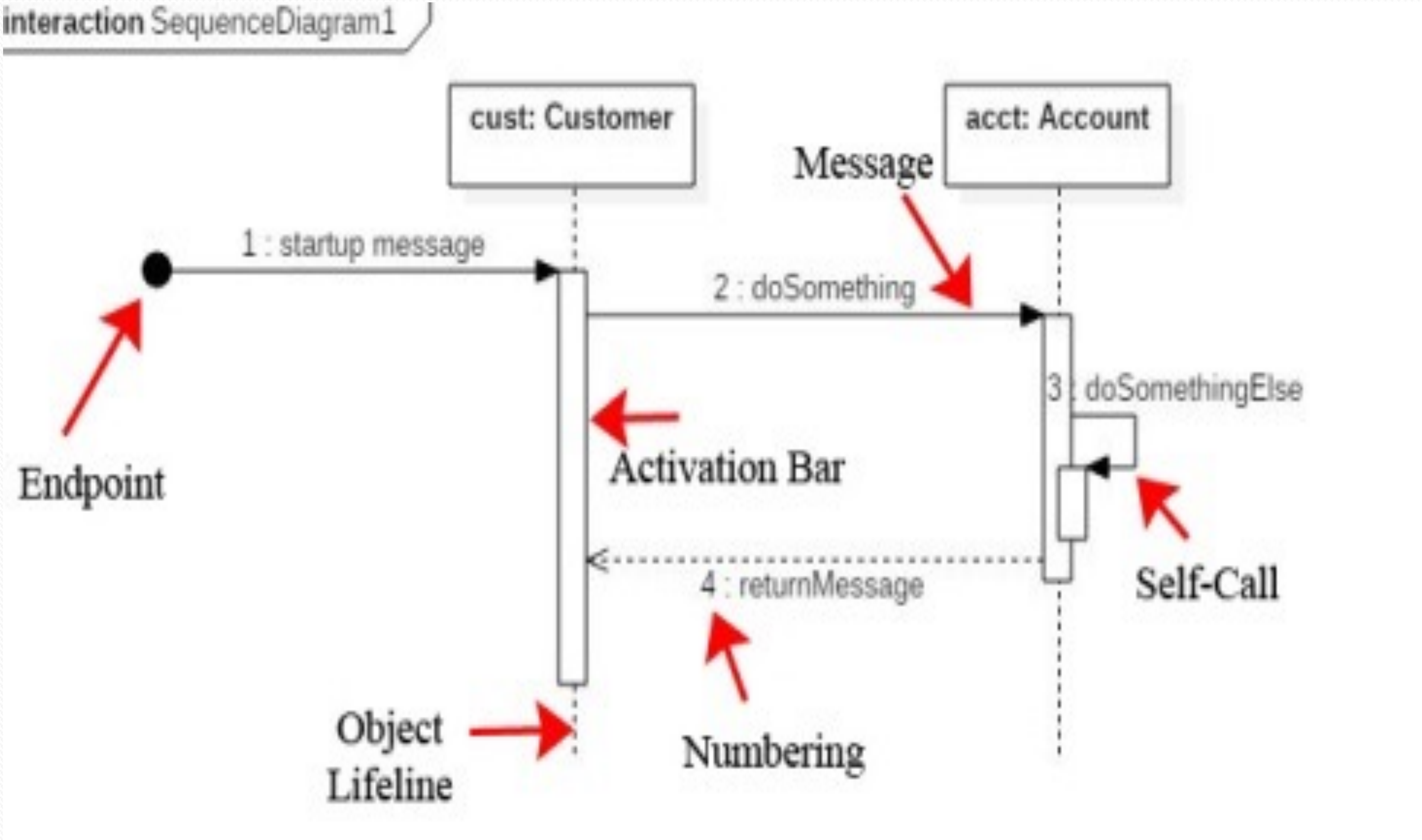
- **Interaction diagrams** describe how groups of objects collaborate in some behavior.
- The UML defines several forms of interaction diagram, of which the most common is the sequence diagram and the other one is collaboration diagram.
- Typically, a sequence diagram captures the behavior of a single scenario of a use case (like “deposit money”, “open account”, “calculate total price of an order”).
- A Sequence Diagram shows the flow of communication

between the running objects of the system, driven by the use case of the system.

# Interaction Diagrams: Overview

- **Sequence Diagrams**
- Object Diagrams
- Delegation
- Polymorphism

# General Syntax



# Sequence Diagrams

A sequence diagram shows interaction between objects

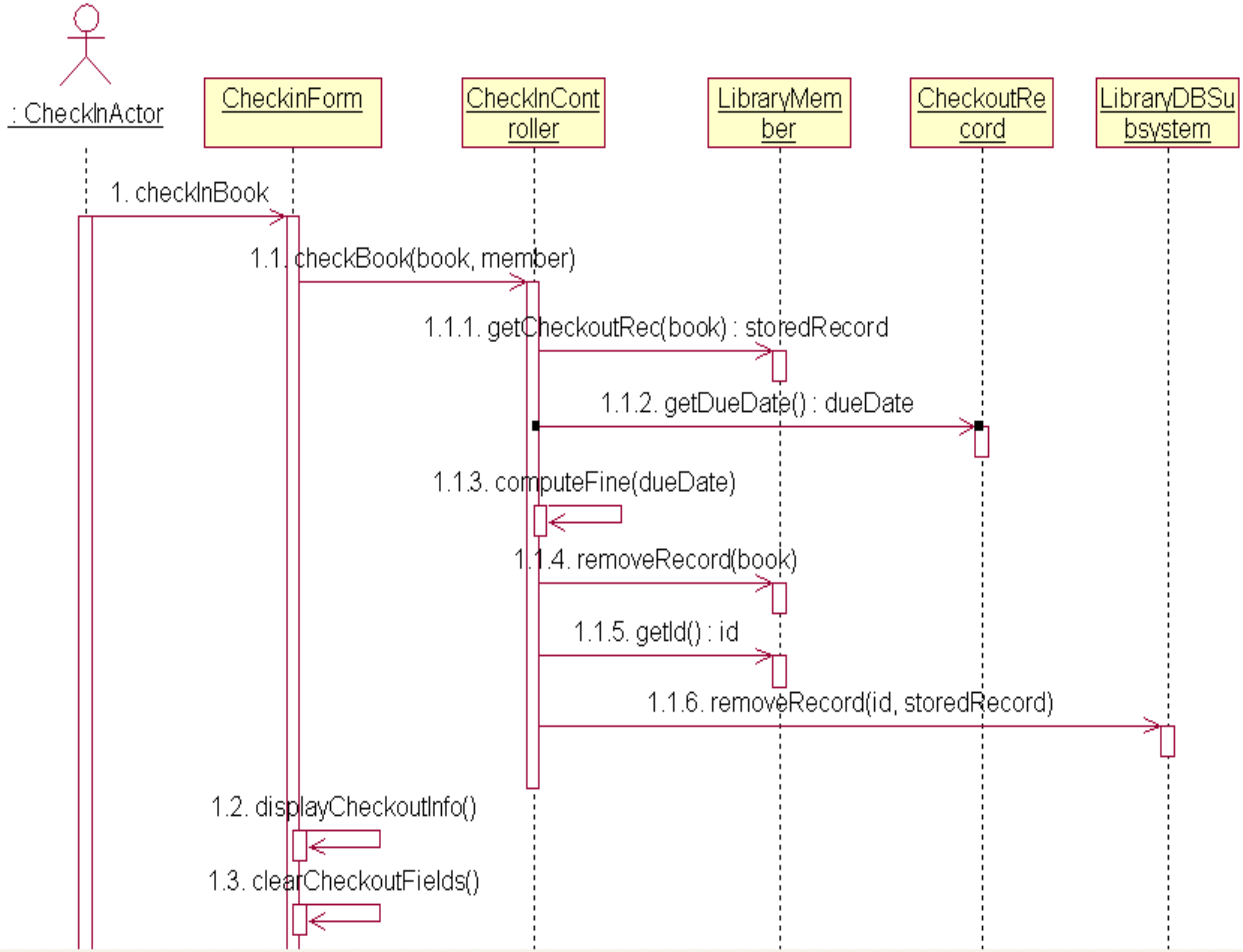
- Horizontal arrows (= messages) indicate calls
- Every message has a number and a name
- Sometimes numbering is *hierarchical* (not used in this course but illustrated in an upcoming slide)
- Activation bars indicate method call duration
- Vertical dotted line shows lifetime of object
- Typically, a sequence diagram begins with an action by an Actor, but sometimes an action may be initiated by some other part of the system. In that case the starting point is called an *Endpoint*. Subsequent steps occur as one object after another is accessed to accomplish the actor's request

# Sequence Diagram Exercise
## (capture from code)

Create a sequence diagram based on the flow that occurs when an actor invokes the `checkinBook` method on `CheckinForm`. You must use hierarchical numbering and activation bars (in the correct way!). You must show all parameters.

```
//FROM CLASS CheckinForm
public void checkinBook() {
   theCheckinController.checkBook(m_book, m_member);
   displayCheckoutInfo();
   clearCheckoutFields();
}

//FROM CLASS CheckinController
public void checkBook(Book book, LibraryMember member) {
   CheckoutRecord storedRecord = member.getCheckoutRec(book);
   Date dueDate = storedRecord.getDueDate();
   double fine = computeFine(dueDate);
   member.removeRecord(book);
   libraryDBSubsys.removeRecord(member.getID(), storedRecord);
}
```

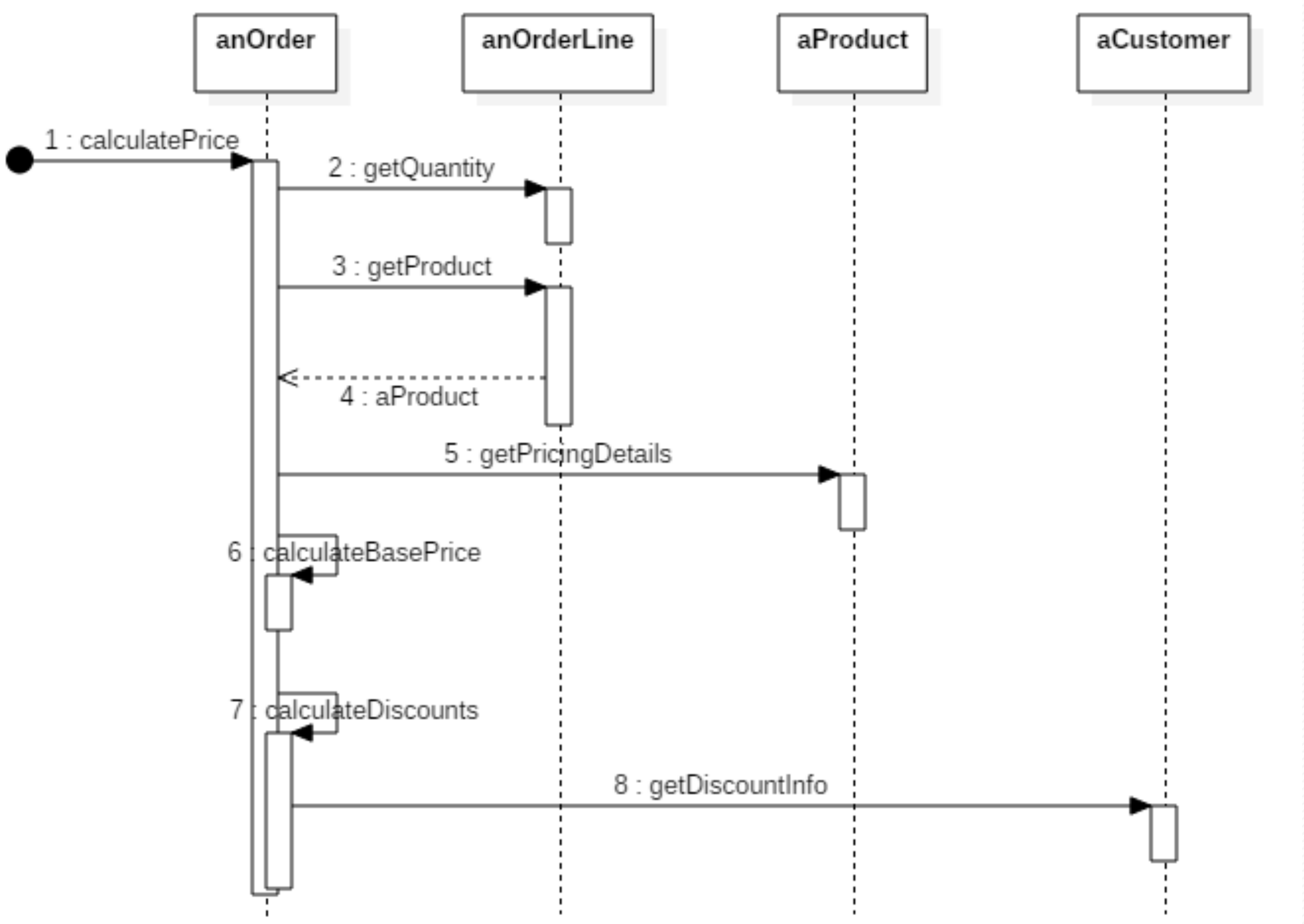
# Solution


: CheckInActor
CheckinForm
CheckInController
LibraryMember
CheckoutRecord
LibraryDBSubsystem
1. checkInBook
1.1. checkBook(book, member)
1.1.1. getCheckoutRec(book) : storedRecord
1.1.2. getDueDate() : dueDate
1.1.3. computeFine(dueDate)
1.1.4. removeRecord(book)
1.1.5. getId() : id
1.1.6. removeRecord(id, storedRecord)
1.2. displayCheckoutInfo()
1.3. clearCheckoutFields()

# Another Example

- We have an order and we are going to invoke a command on it to calculate its price.
- To do that, the order needs to look at all the line items on the order and determine their prices, which are based on the pricing rules of the order line's products.
- Having done that for all the line items, the order then needs to compute an overall discount, which is based on rules tied to the customer.

# Centralized Control Solution



# About the Diagram

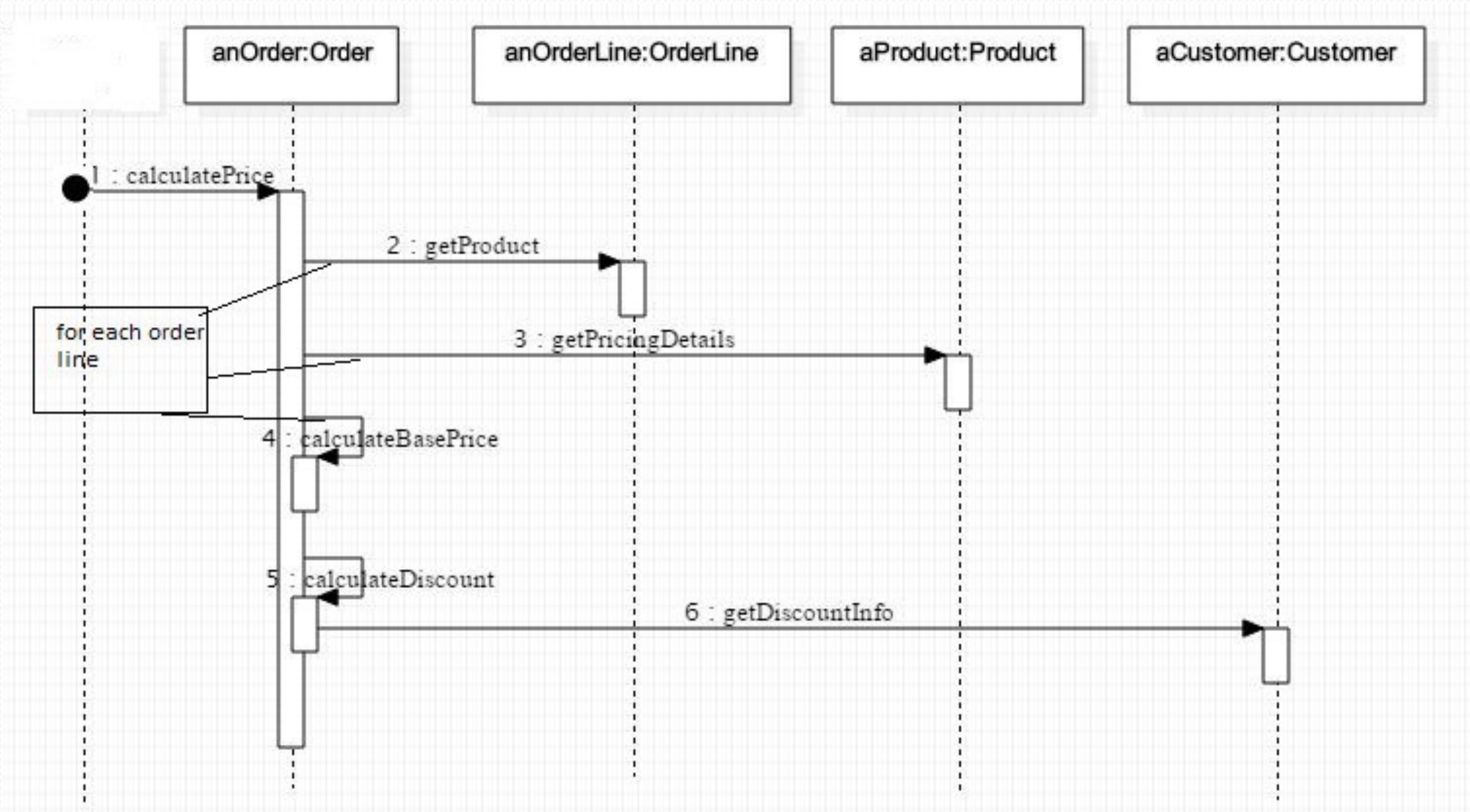
- Activation bars. These always mark the beginning and end of a “method call”
- Endpoint instead of Actor. In modeling real systems, actions are typically initiated by an Actor, but to display the behavior of a portion of a system, an action may be initiated by an *Endpoint*, as shown here.
- Centralized Control. Solutions in which there is one primary controlling class are often used for understanding a problem domain, but during design, control is typically *distributed* so that different objects handle different parts of the flow according to their responsibilities

- <u>Indicating Loops</u>: The sequence of messages getQuantity, getProduct, getPricingDetails, and calculateBasePrice needs to be done for each order line on the order, while calculateDiscounts is invoked just once. Diagram does not show that these loops are occurring.
- <u>How to show looping is occurring</u>. There are several UML ways to do this:
    - *Use Notes*. To indicate an operation is repeated, a simple note can be used.

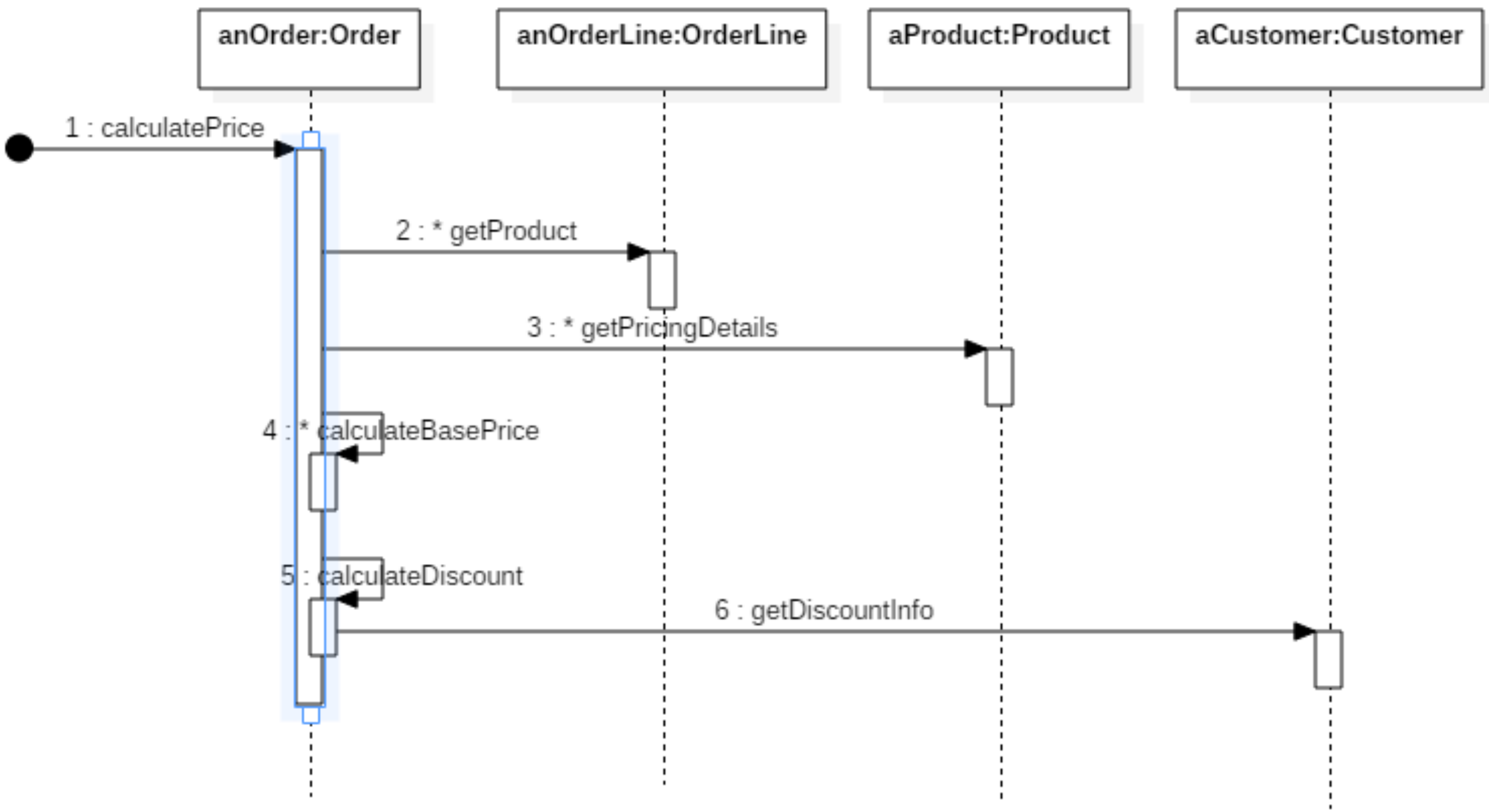
    - *Use an Iteration Marker*. Marking an operation with an asterisk (*) indicates that the operation repeats. This is an economical way, but gives no information other than a loop is occurring. [UML2 considers this approach to be deprecated – but it is still used sometimes anyway]

    - *Use an Interaction Frame*. Introduced in UML2.0. An interaction frame marks off a piece of a sequence diagram to indicate that a loop is occurring there. One objection to this approach is that it makes the diagram harder to read – in this course we do not use interaction frames.

# Showing Looping with Notes



# Showing Looping with Iteration Marker



# About These Diagrams

- *Loops*. The previous diagrams show two ways to indicate looping – in this course we will use iteration markers
- *Representing Objects in UML.* Diagrams also show proper UML syntax for indicating objects along the top. The syntax is:

  *instanceName : className*

  Both can be included, or one or the other can be dropped

# Distributed Control Solution

# Notes About Distributed Solution

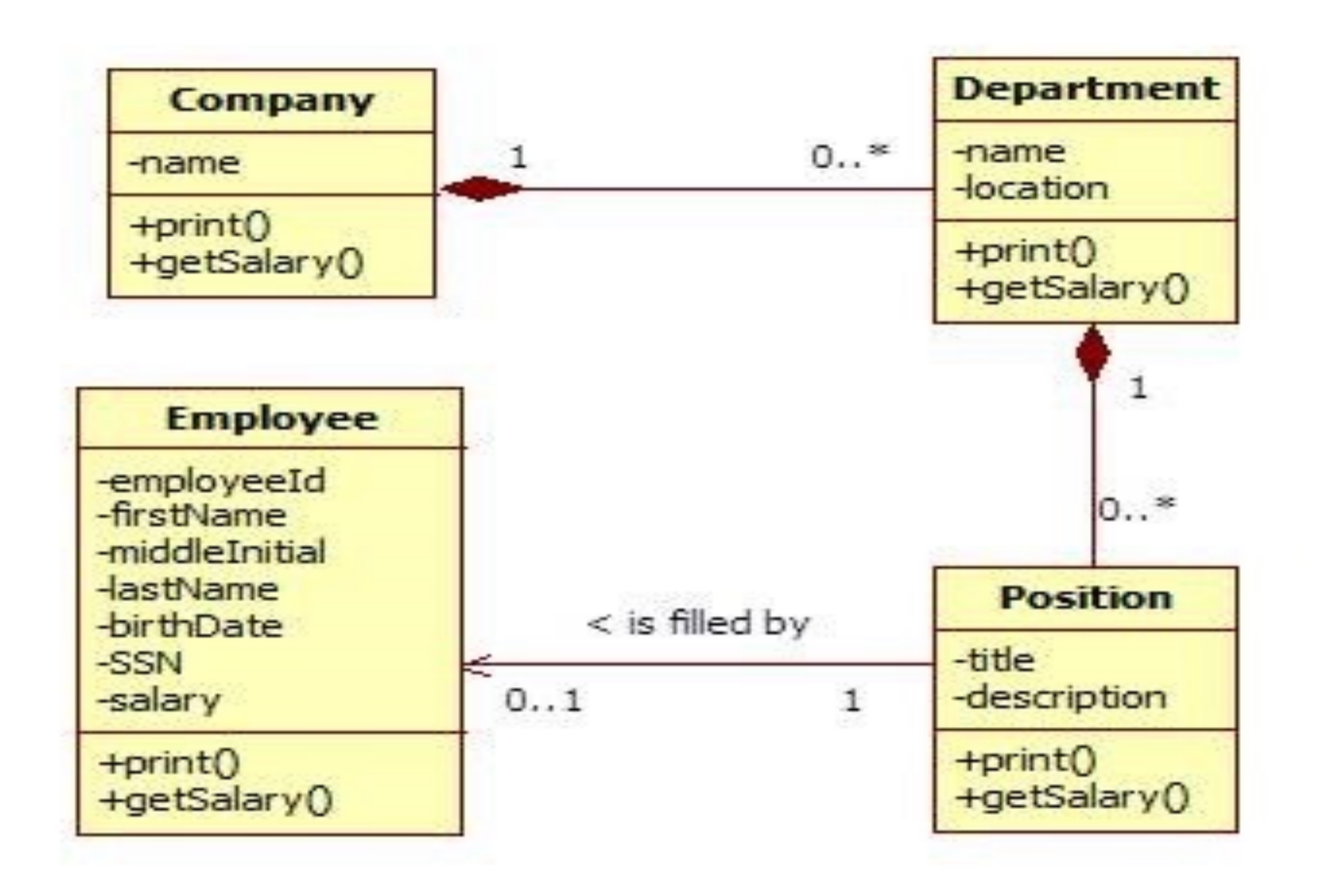
- In the distributed solution, steps to achieve the goal of the use case are carried out by objects that are designed to do those activities
- Since Order is a top level object in this use case, it is not reasonable to expect it to know all the details about adding up prices at a fine-grained level, but it is reasonable to expect it to know where to get that information. So Order delegates the responsibility of calculating price for a line item to OrderLine objects, obtaining the base value.
- After base value has been computed, Order asks the Customer object to factor in its discount to compute total price, which is returned at the end.

# Exercise

- A Company has a name and many Departments, each department has a name, location, and many Positions. Each position has a title and a description, and is filled by a single Employee, which has an employeeId, firstname, middleInitial, lastName, birthDate, SSN, and Salary

- <u>The Task</u>: Create a class diagram to provide a model of the above description.

# The Class Diagram



# The Code – the Classes with Attributes

TASK: Write a program that makes a call to Company to compute the total of all salaries of Employees in the Company. First code the classes with attributes, then create a sequence diagram to model the flow

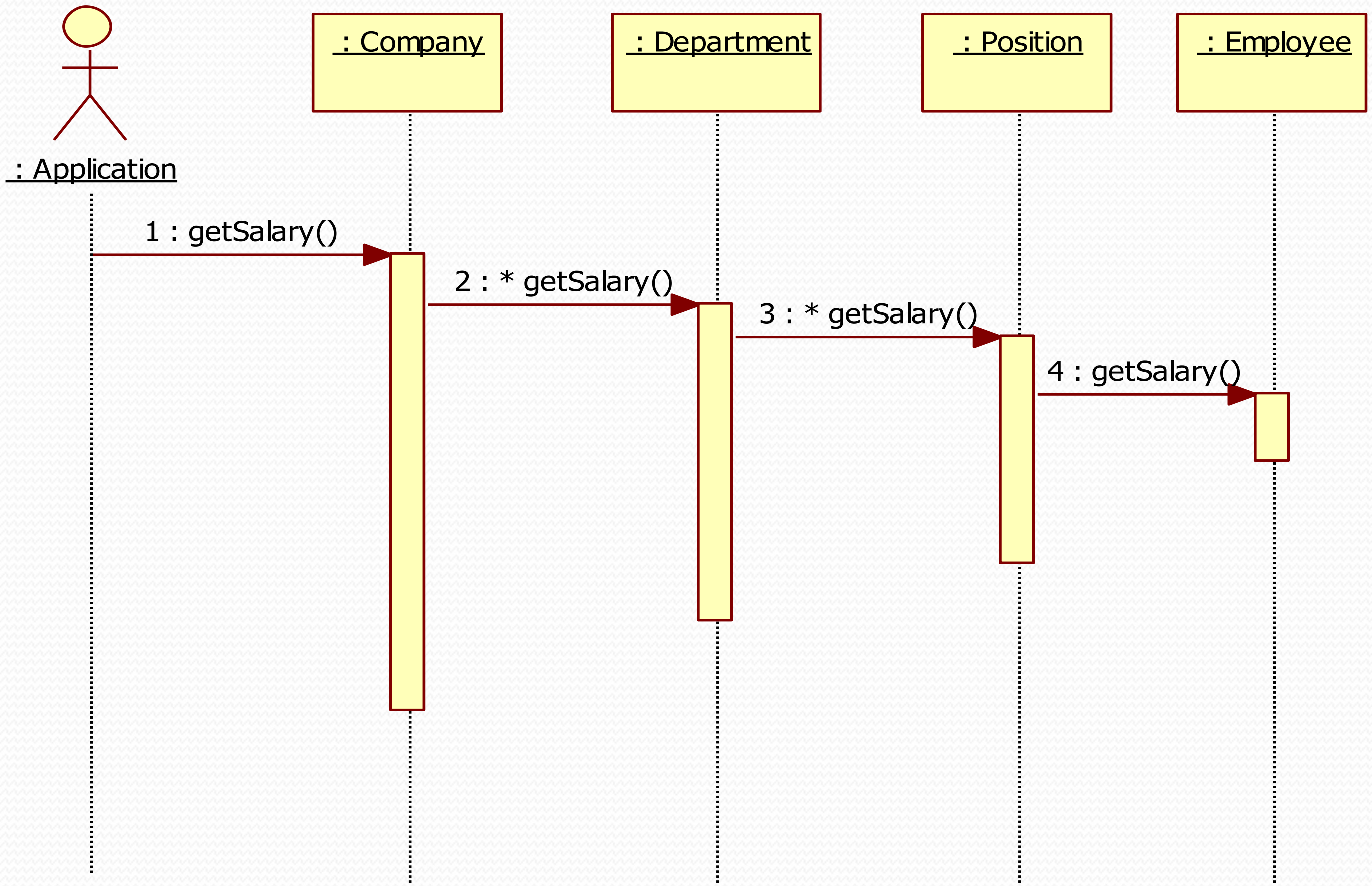
```
public class Company {
  private String name;
  private List<Department> departments;

}
```

```
public class Department {
  private String name;
  private String location;
  private List<Position> positions;
}
```

```
public class Employee {
  private String employeeId;
  private String firstName;
  private String middleInitial;
  private String lastName;
  private String SSN;
  private Date birthDate;
  private double salary;

}
```

```
public class Position {
  private String title;
  private String description;
  private Employee emp;

}
```

# A Distributed Control Solution




**Note**: Here, the responsibility of computing salary is *delegated* to more and more fine-grained objects that carry out the task according to their own level of responsibility. Ultimately, the computation depends on each Employee salary. It is obvious and natural for control to be *distributed* in this case.

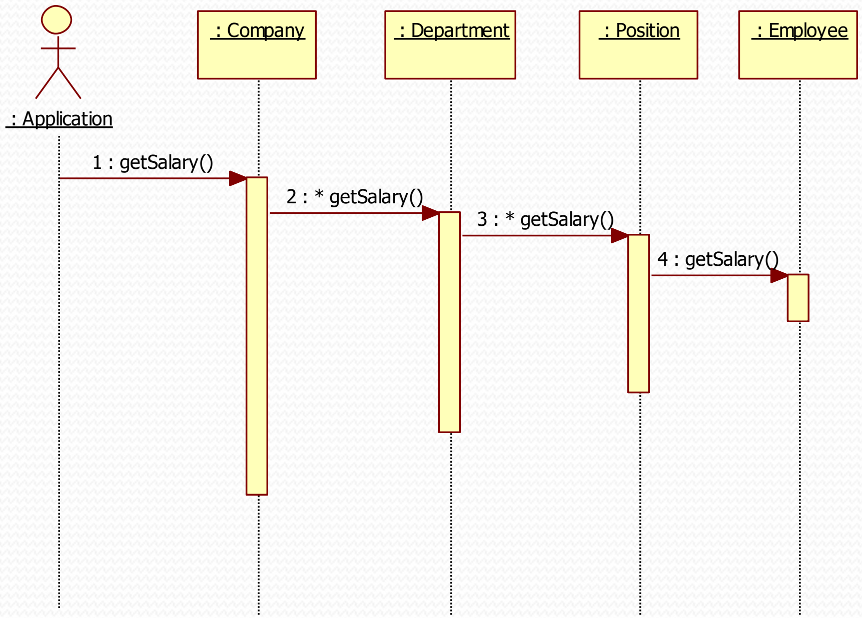
# Turning It into Code

- Now we add the code showing the methods we will use to print out the salaries in our four classes.
- Here is the simple main class.

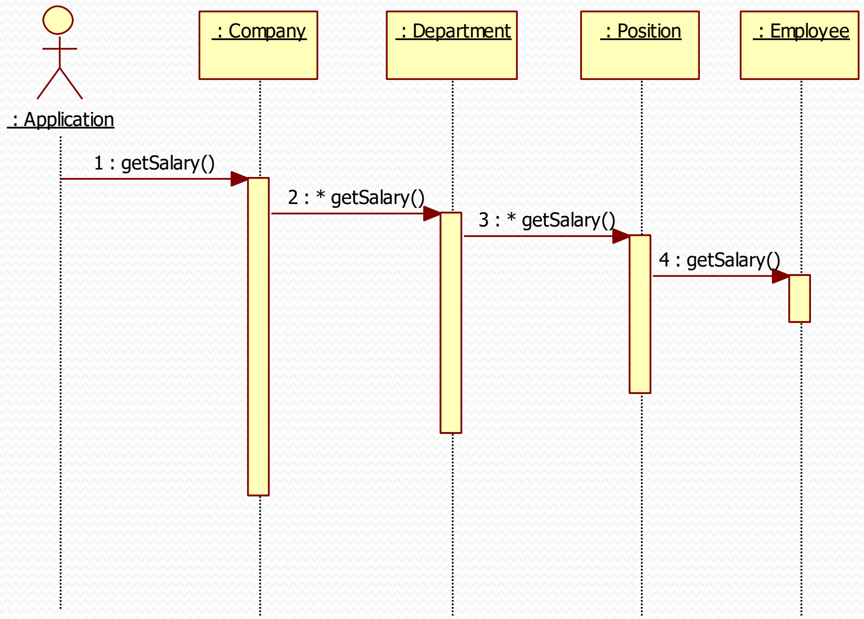
```
public class Application {
  public static void main(String[] args) {
    ...
    double totalSalary = company.getSalary();
  }
}
```

# (continued)

```
public class Company {
  private String name;
  private List<Department> departments;

  public double getSalary() {
    double result = 0.0;
    for (Department dep : departments) {
      result += dep.getSalary();
    }
    return result;
  }
}
```



```java
public class Department {
  private String name;
  private String location;
  private List<Position> positions;

  public double getSalary() {
    double result = 0.0;
    for (Position p : positions) {
      result += p.getSalary();
    }
    return result;
  }
}
```




```java
public class Position {
  private String title;
  private String description;
  private Employee emp;

  public double getSalary() {
    return emp.getSalary();
  }
}
```

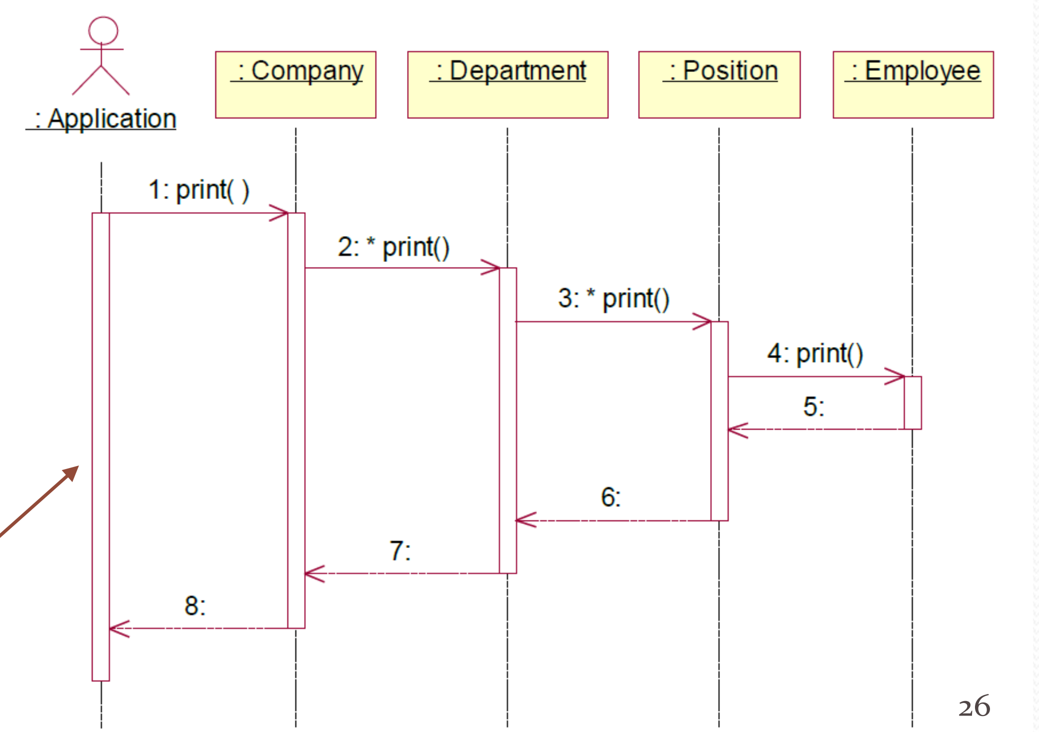
```java
public class Employee {
  private String firstname;
  private double salary;

  public double getSalary() {
    return salary;
  }
}
```

# Return Arrows

- Return arrows may be shown, optionally
- When a return value makes the diagram more understandable, it is good to show it
- Showing all return arrows is bad practice because it clutters the diagram

| |
|---|
| Sometimes a Note is used to indicate an important return value |
| The example here shows what a waste of space return arrows can be – and they do not add anything valuable |

**Bad use of returns**



# Main Point 1

Sequence Diagrams document the sequence of calls different objects (should) make to accomplish a specific task.

Likewise, harmony exists in diversity: Even though each object is specialized to only perform tasks related to itself, objects harmoniously collaborate to create functionality far beyond each object's individual scope.

# Quiz - 1

- In sequence diagram which one of the following mark the beginning and end of a “method call”.
  a) Actor
  b) Activation Bar
  c) Horizontal Arrow

# Interaction Diagrams: Overview

- Sequence Diagrams
- **Object Diagrams**
- Delegation and Propagation
- Polymorphism

# Object Diagrams

- An **object diagram** is a snapshot of the objects in a system at a point in time.
- Because it shows instances rather than classes, an object diagram is often called an *instance diagram*.

Object diagrams look superficially similar to class diagrams however the boxes represent specific instances of objects.

Boxes are titled with :-

objectName : ClassName

As each box describes a particular object at a specific moment in time the box contains attributes and their values (at that moment in time).

attribute = value

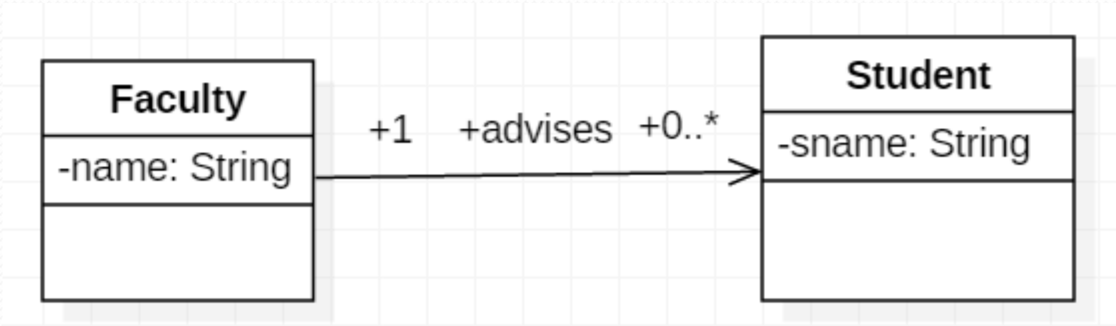
These diagrams are useful for illustrating particular 'snapshot' scenarios during design.

# Object Diagram Syntax

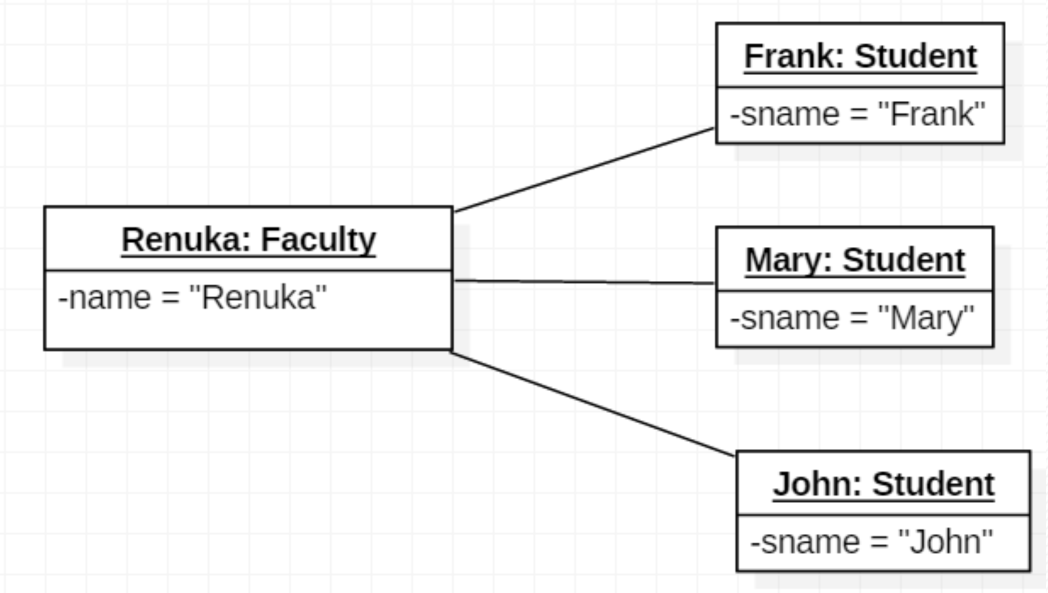
- Underlining indicates it’s an object
- Usually shows colon separated name and type
- Associations in object diagrams don’t have multiplicities
- Associations may or may not display an arrowhead
- The state of each object is shown

# Example

## Class Diagram




**Draw the Object diagram. Faculty Renuka is advising to three students.**



# Main Point 2

Object Diagrams show the relationships between objects, where each object is an instance of a class, and each reference is represented by a single arrow or line.

This phenomenon illustrates the principle that *the whole is greater than the sum of the parts*: The objects (parts) on their own are not the important focus for an object diagram. What is important is how the objects relate; together, objects and their relationships form a whole that is more than just the sum of individual objects collected together.

# Quiz - 2

Object diagrams don't have multiplicities.

a) True

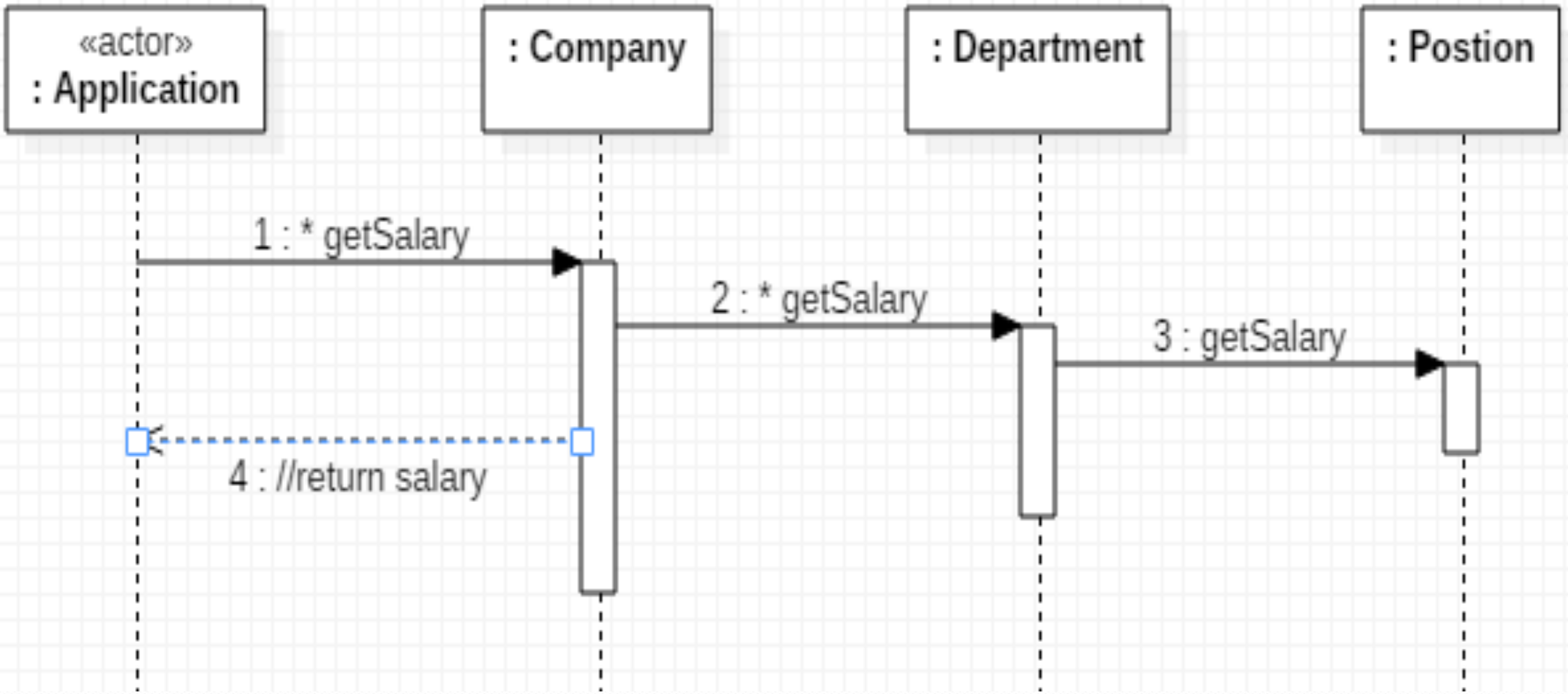
b) False

# Interaction Diagrams: Overview

- Sequence Diagrams
- Object Diagrams
- **Delegation**
- Polymorphism

# Delegation

- A class can express functionality in its interface, but it may delegate some or all of the responsibility to an associated class to carry out the action
- The responsibility for the action can propagate through a hierarchy.



# Main Point 3

OO Systems use delegation. An individual object only works with its own properties, acts only on what it knows, and then asks related objects to do what they know.

When individual actions are on the basis of self-referral dynamics, individual actions are automatically in harmony with each other because all arise from the dynamics of the a single unified field.

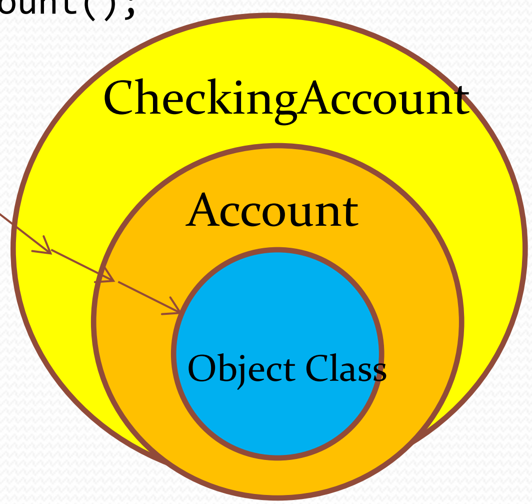
# Interaction Diagrams: Overview

- Sequence Diagrams
- Object Diagrams
- Delegation and Propagation
- **Polymorphism**

# Polymorphism

- Polymorphism = many forms
  - Objects of a particular type can take different forms
  - Achieved through dynamic binding (late binding)
  - Implies that a type has subtypes (extends, implements)

```
Account act = new CheckingAccount();
act.toString();
```




```
Account[] accts =
   {new CheckingAccount(),
    new SavingsAccount()};
double total = 0.0;
for(Account a: accts) {
   total += a.getBalance();
}
```

The method call first checks the class of the actual object to find a `toString()`method; if not found, it checks successive super classes, rising finally to Object

The runtime first checks the runtime type of the object to find a `getBalance()` method; if not found, it checks successive super classes, rising finally to Object

# Late Binding

- Binding is the connection of a method call to a method implementation.

- Late binding, or dynamic binding, occurs at run-time.
  - the JVM method-call mechanism finds the correct method body and invokes it at run-time.
    - by traversing the inheritance chain, starting at the runtime type of the object
  - late binding is the implementation mechanism that makes polymorphism work

# Early Binding

Static, private, final methods are bound to the correct method body at compile time – this is called *early binding.*

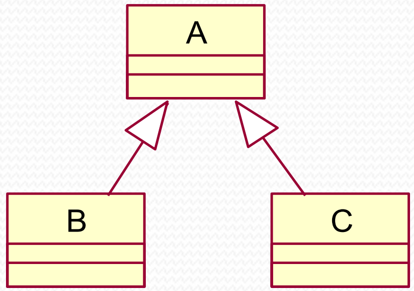
**<u>Static methods</u>**. When a call is made to a static method, the method body may not be in the current class – it may be in a super class or some more distant ancestor. The compiler will climb the inheritance chain till it finds the first occurrence of an implemented version of the method and creates the binding – see demo lesson4.lecture.staticinherit.fifth

**<u>Private methods</u>**. When a private method is called on an object of type A, there is no possibility it was overridden in a subclass, and because of the visibility rules for overriding, it could not have been inherited from a superclass. The binding is uniquely determined in this case

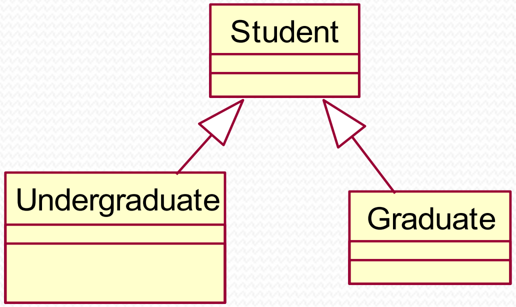
**<u>Final methods</u>**. When a final method is called, it could not have been overridden in a subclass. The compiler climbs the inheritance chain till it finds the first place where this method has an implementation and performs the binding to that one. See demo lesson4.lecture.finalinherit

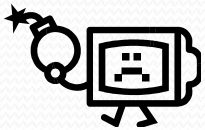
# Static Typing

Objects of type B or C can be declared as type A




```java
public class Student { ... }
public class Undergraduate extends Student { ... }
public class Graduate extends Student { ... }

Student st1, st2, st3;
Graduate st4;
st1 = new Student();
st2 = new Undergraduate();
st3 = new Graduate();
st4 = new Student();
```

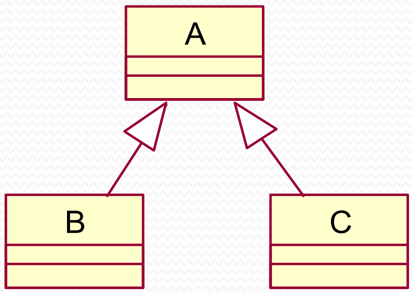


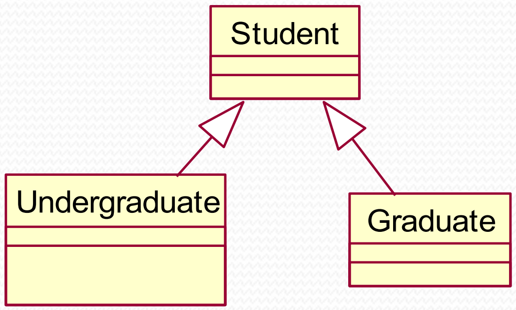
Where is the Compiler Error?

# (continued)

Objects of type B or C can be cast as type A




```
public class Student { ... }
public class Undergraduate extends Student { ... }
public class Graduate extends Student { ... }

Student st1, st2, st3;
Graduate st4;
st1 = new Student();
st2 = new Undergraduate();
st3 = new Graduate();
st4 = new Student(); //error
```



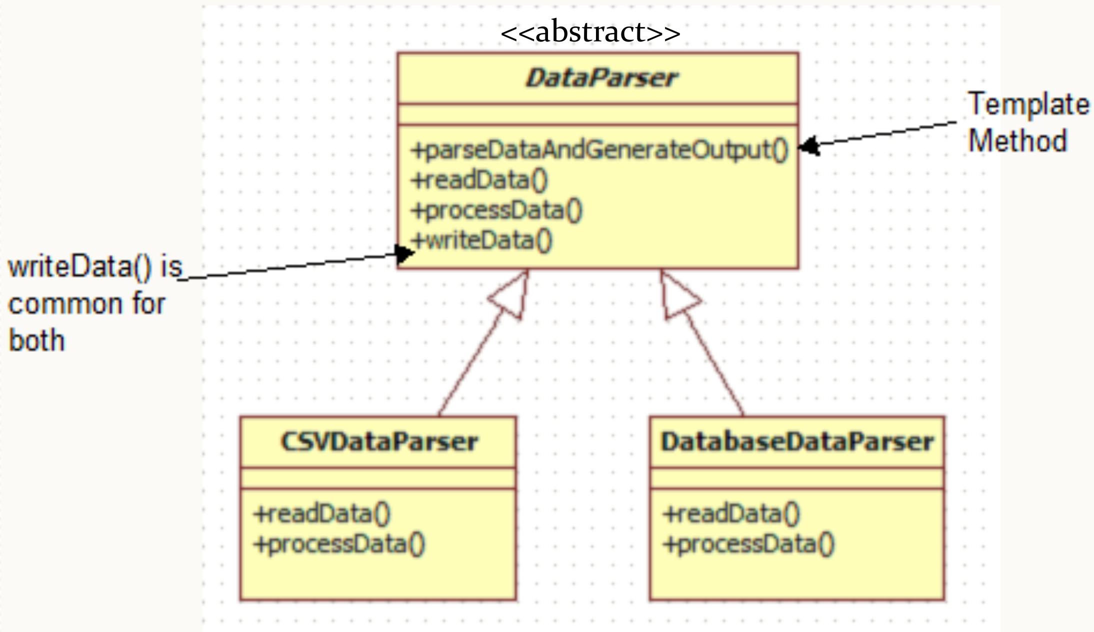

NOTE: In the first three cases, the Student class is the *static type* of the object created. The *runtime types* are, respectively, `Student`, `Undergraduate`, and `Graduate`.

# Open-Closed Principle

- Software should be designed so that it is **open for extension**, but **closed for modification**.
- All systems change during their life cycle, but when a single change results in a cascade of changes, the program becomes fragile and unpredictable. When requirements change, you implement these changes by adding new code, not by changing old code that already works.
- Demo: lesson4.lecture.openclosed.closedcurve
- Software modules can never be 100% closed for modification. Programmer has to decide what aspects should be closed.

# Polymorphism and the Template Method Pattern

Sometimes a class at the top of an inheritance hierarchy needs to carry out a sequence of tasks, some of which need to be implemented by subclasses. This situation is an example of the *Template Method Design Pattern* – see lesson4.lecture.template

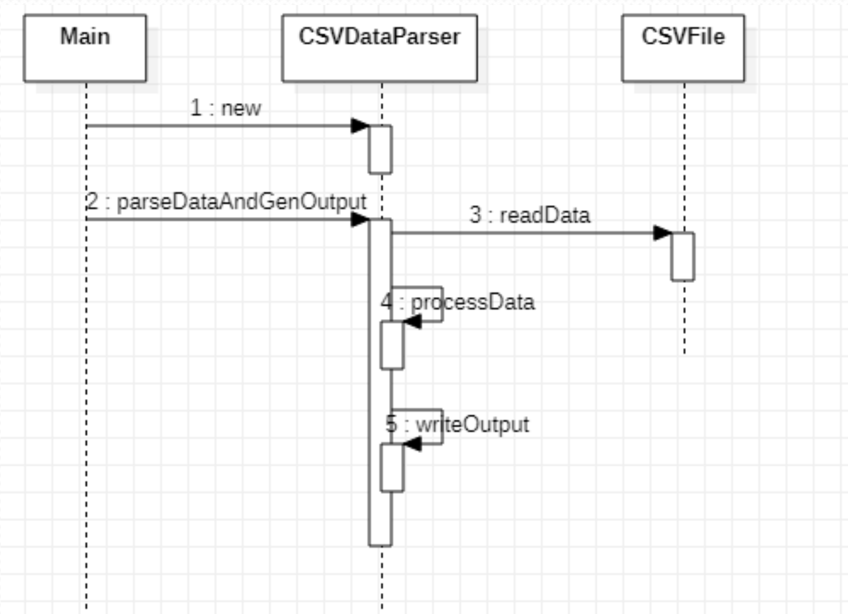



```
public void parseDataAndGenerateOutput() {
    readData();
    processData();
    writeData();
}
```

# Sequence Diagrams When Inheritance is Present

<u>Basic Principle</u>: Sequence diagrams are concerned with runtime objects, so, generally speaking, superclasses do not need to appear in the sequence diagram (though there can be exceptions).

<u>Example from Previous Slide</u>: Consider scenario when we wish to parse CSV data.




*No need to mention the DataParser superclass in this diagram*

# Main Point 4

Polymorphism supports use of the *Open-Closed Principle*: The part of our code that is established and tested is closed to modification (change), but at the same time the system remains open to changes, in the form of *extensions*.

In a similar way, progress in life is vitally important, and progress requires continual change and adaptation. But change stops being progressive if it undermines the integrity of life. Adaptability must be on the ground of stability.

# Quiz - 3

- Which one of the following method uses early binding?
- A) private
- B) static
- C) final
- D) all the above

# Summary

Today we looked at modeling Object Collaboration and the uses of Polymorphism.

- Sequence diagrams document the sequence of method calls between objects
- Object diagrams show the relationships between objects. It is important to know how a class diagram translates into an Object Diagram
- The OO tools of association, delegation, and polymorphism allow us to build software solutions that reflect accurately the system we are modeling and are efficient, flexible and extensible.

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Sequence Diagrams and Object Diagrams both show how objects relate to each other.

2. To preserve encapsulation, objects should only act on their own properties, and to accomplish tasks that are the responsibility of other objects, they should send messages (delegation)

---

3. **<u>Transcendental Consciousness</u>** by its very nature, has the fundamental association of self-referral – the Self being aware of the Self

4. **<u>Wholeness moving within itself</u>**: In Unity Consciousness one feels intimately associated with all other things in creation as a result of perceiving all things in terms of one's Self