MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS401 Modern Programming Practices (MPP)

Professor Paul Corazza

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 *propagation and 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 *propagated* to other objects; the task is said to be *delegated* to others. 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.

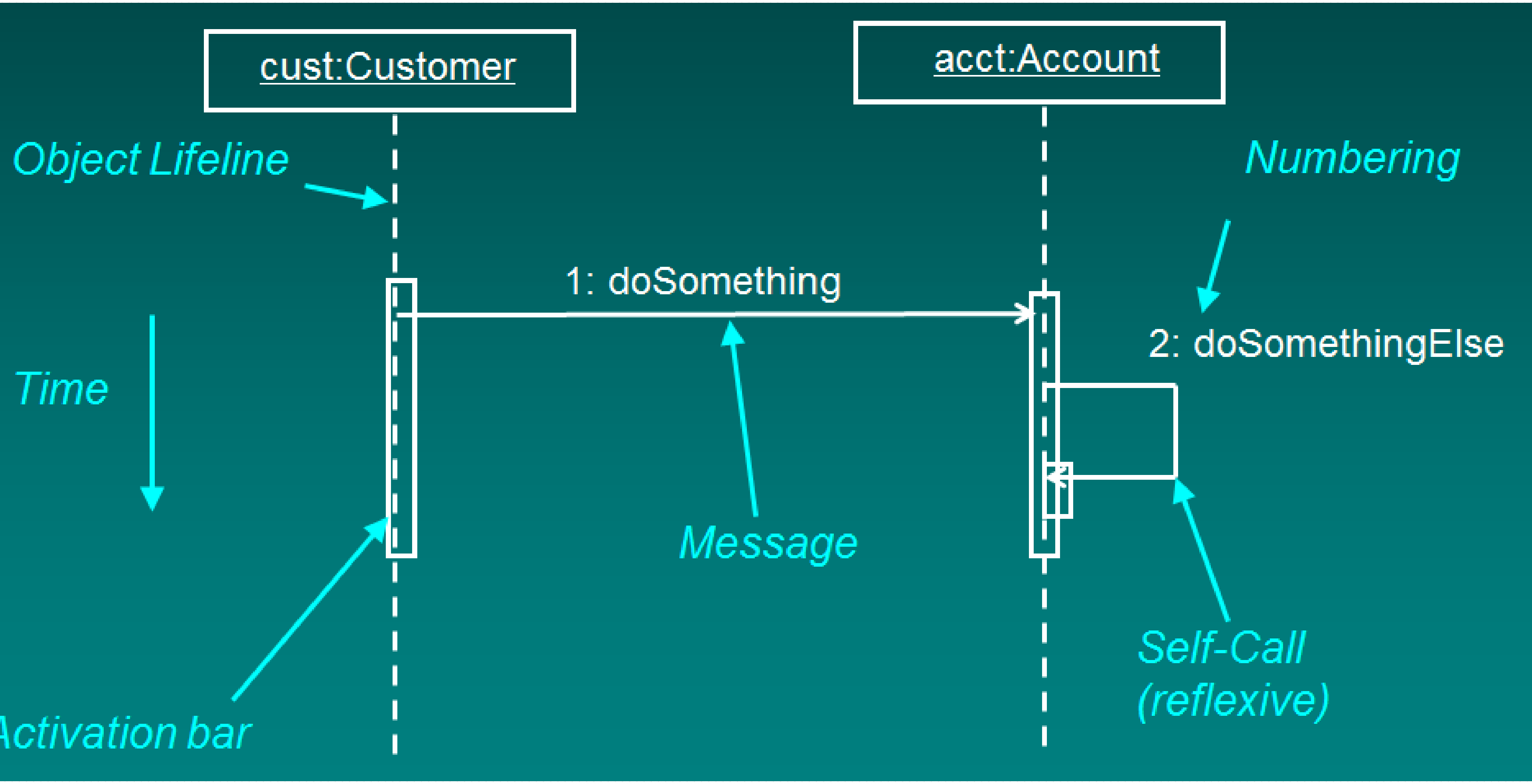
Interaction Diagrams

- **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.
- 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”).
- The diagram shows a number of example objects and the messages that are passed between these objects within the use case.

Interaction Diagrams: Overview

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

Anatomy of Sequence Diagram



Sequence Diagrams

A sequence diagram shows interaction between objects

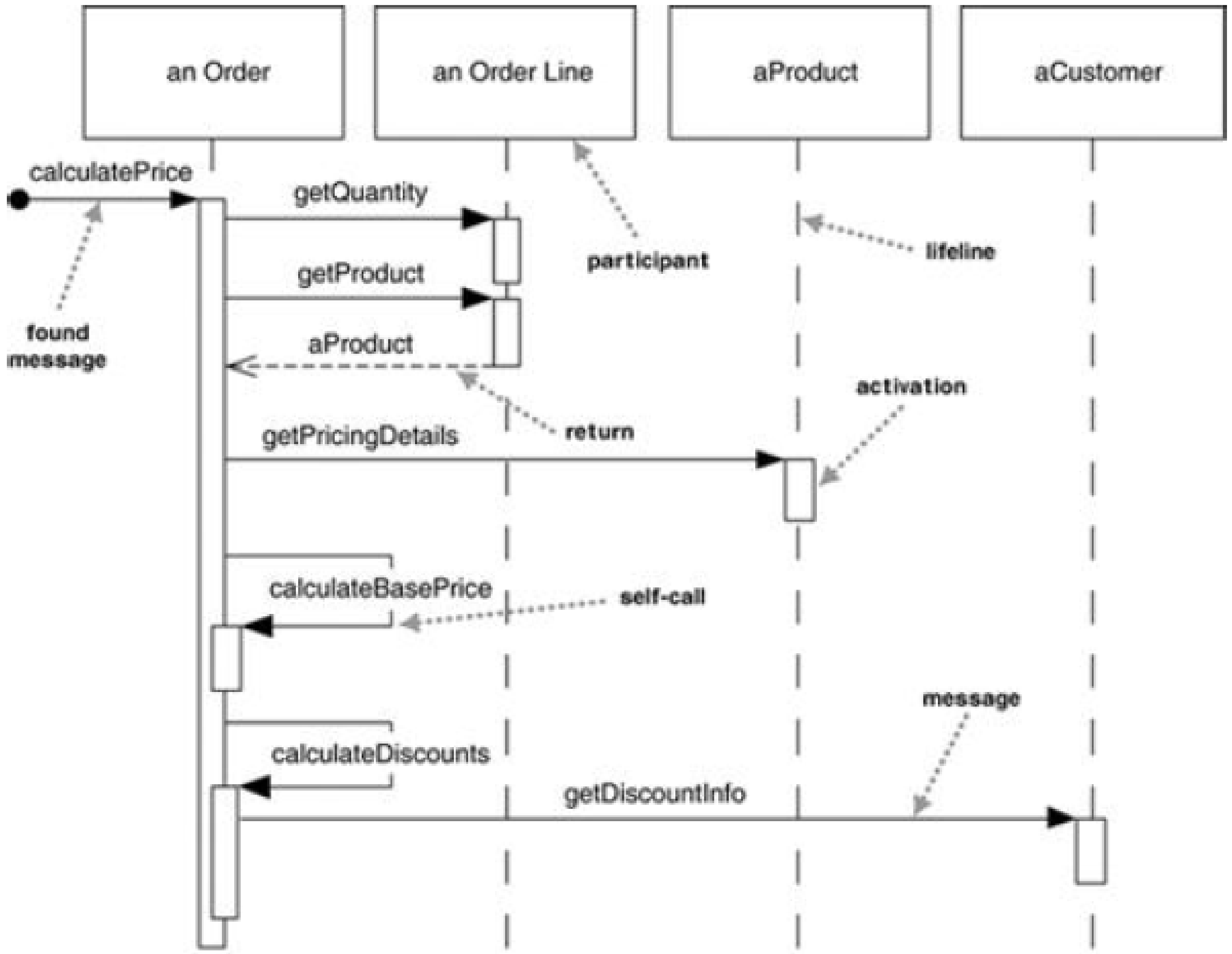
- Horizontal arrows indicate calls (sending messages)
- Every arrow has a number, name, optional multiplicity
- Activation bars indicate method call duration
- Vertical dotted line shows lifetime of object
- Is a dynamic view of a scenario through a Use Case

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

Figure 4.1. A sequence diagram for centralized control

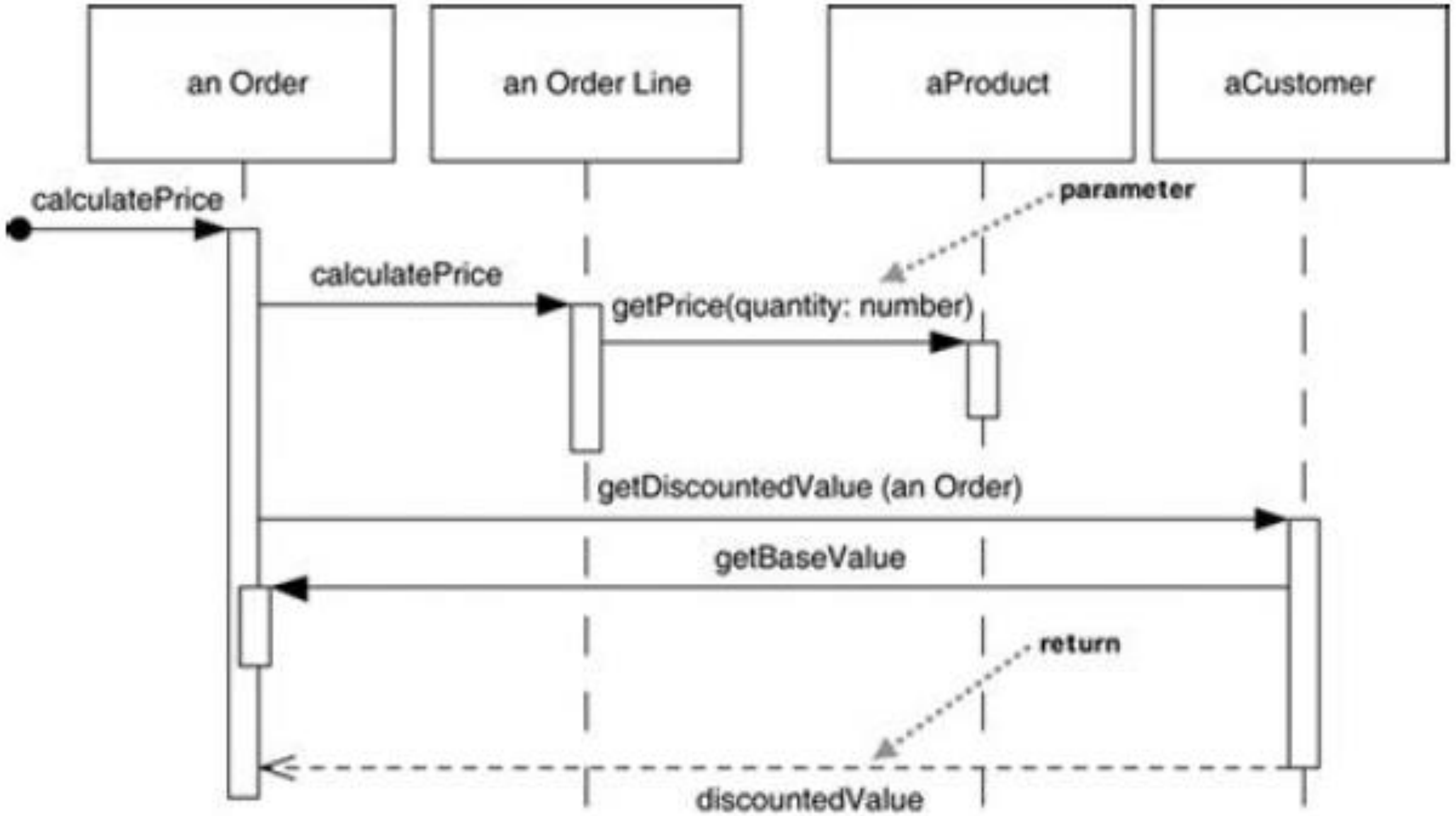


About the Diagram

- Diagram is easy to understand
- Actor not shown. In modeling real systems, actions are always initiated by an actor (could be person or system)
- Not made clear: 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. Could be improved by adding comments.
- Solutions in which there is one primary controlling class are often used for understanding a problem domain, but during design, control is *distributed* so that different objects handle different parts of the scenario according to their responsibilities

Distributed Control Solution

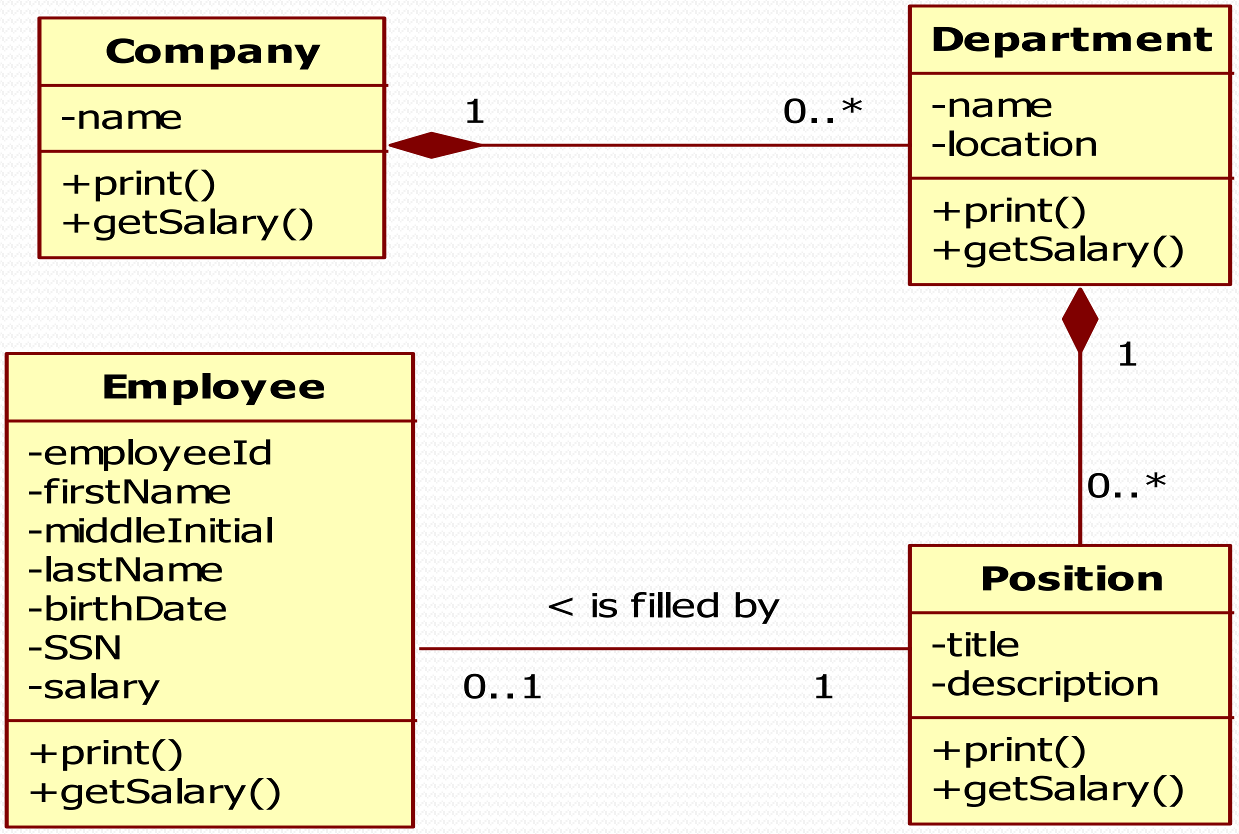
Figure 4.2. A sequence diagram for distributed control



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 fulfilled by a single Employee, which has an employeeId, firstname, middleInitial, lastName, birthDate, SSN, and Salary

The Class Diagram



The Code

```
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;

}
```

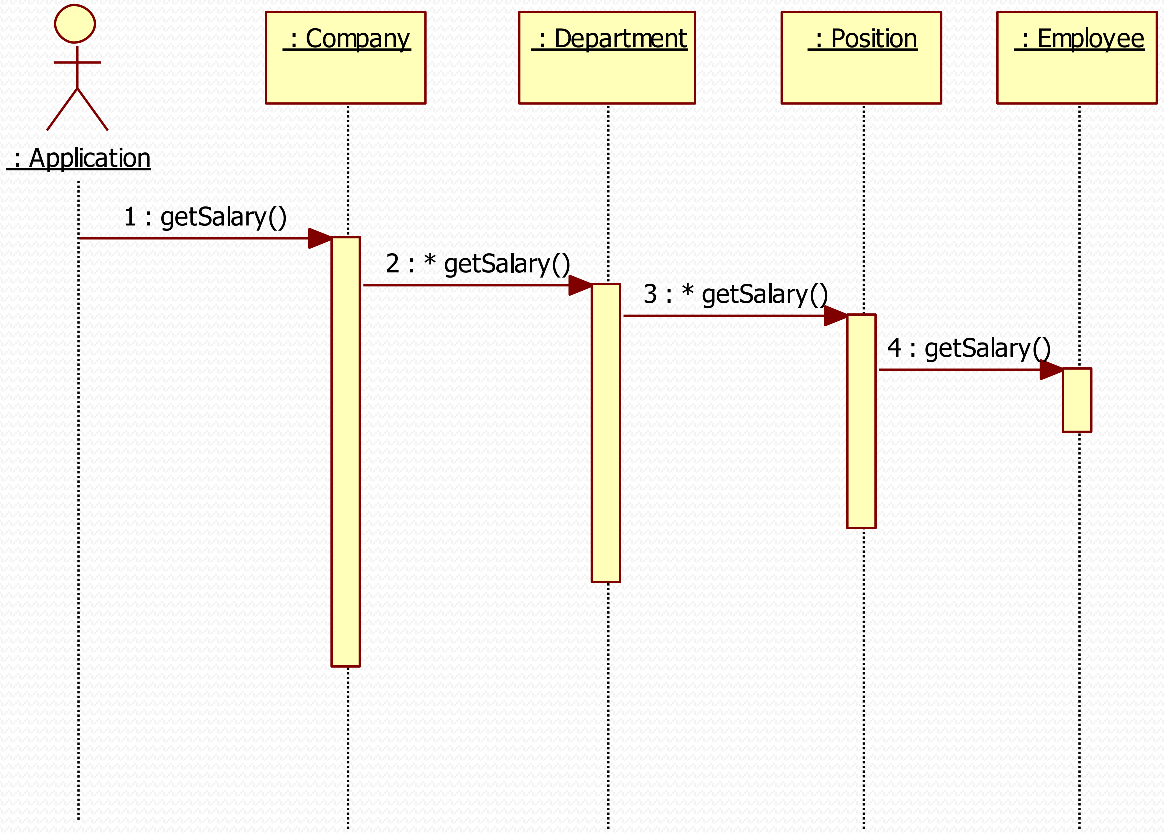
```
public class Position {
  private String title;
  private String description;
  private Employee emp;

}
```

Suppose we want to have multiple employees Fill a position (e.g. senior software developer.) What code change do we make?

The Task

- Draw a sequence diagram showing how the main Application could compute the sum of all the salaries of all the Employees in the Company

A Distributed Control Solution



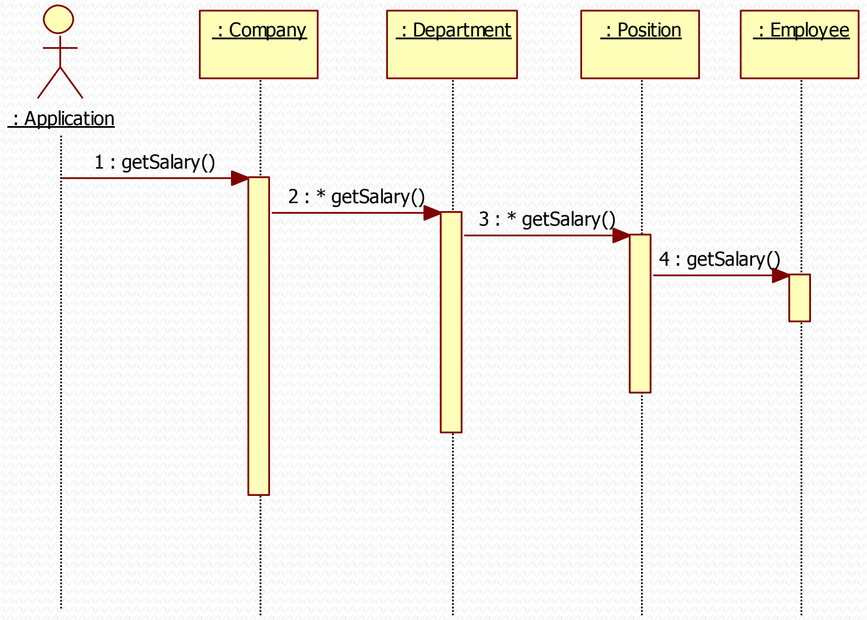
Turning It into Code

- 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.

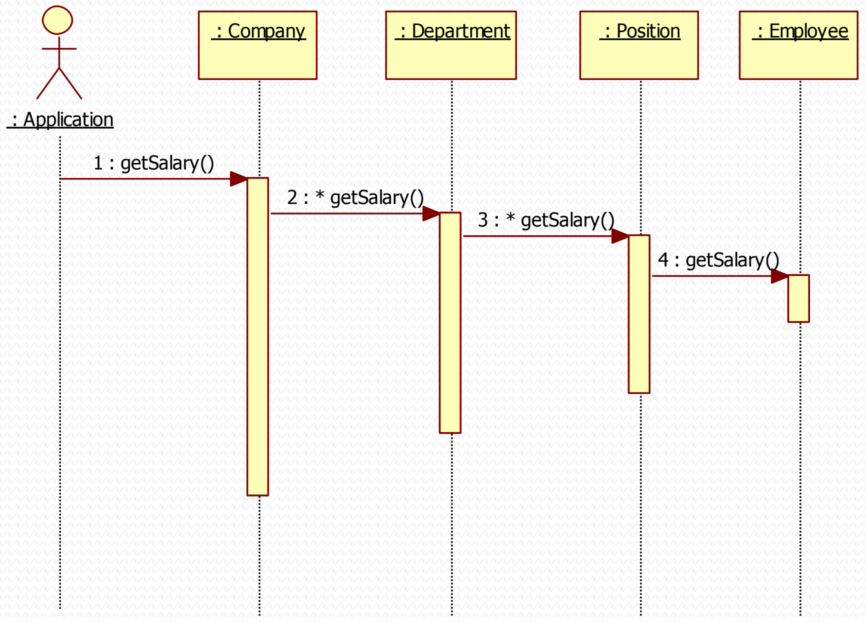
```
public class Application {
  public static void main(String[] args) {
    ...
    double totalSalary = company.getSalary();
  }
}
```

(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;
  }
}
```

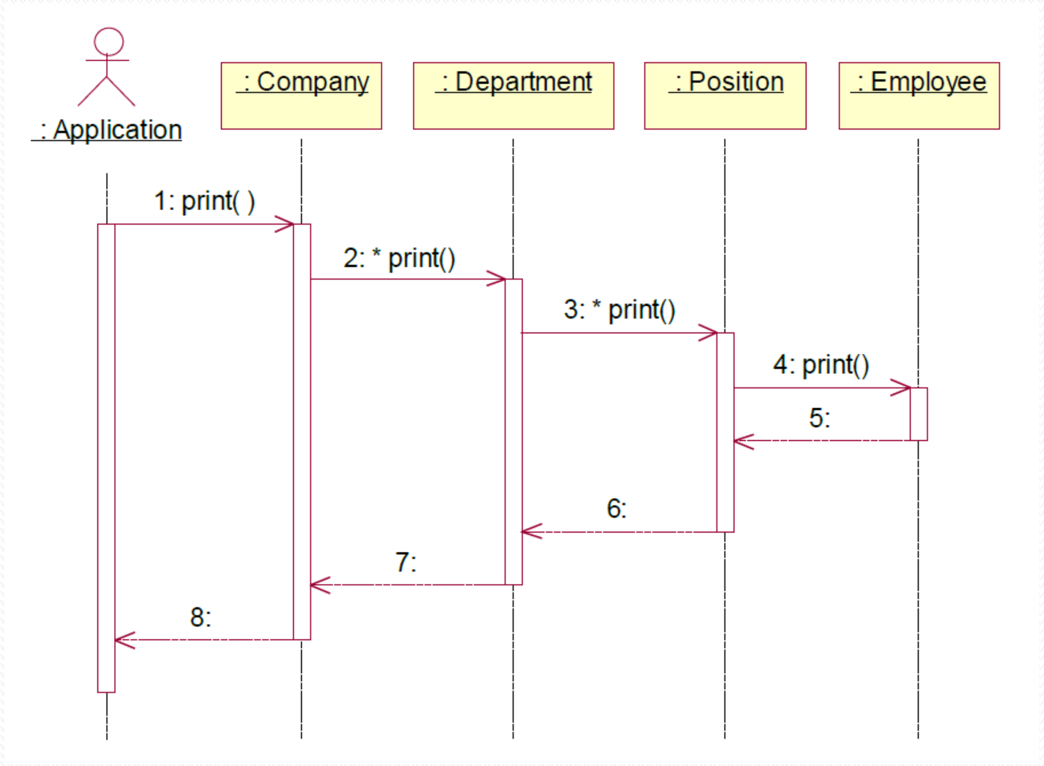


```java
public class Position {
  private String title;
  private String description;
  private Employee emp;

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

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

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

Return Arrows

- Optionally you can also add arrows for returns
- Not all tools support this feature



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.

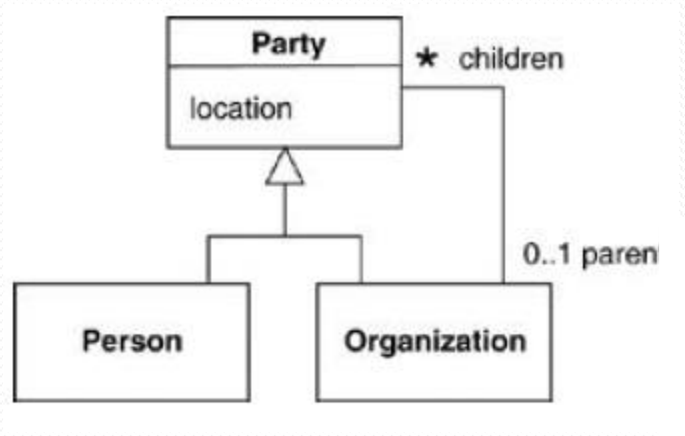
Interaction Diagrams: Overview

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

Object Diagrams

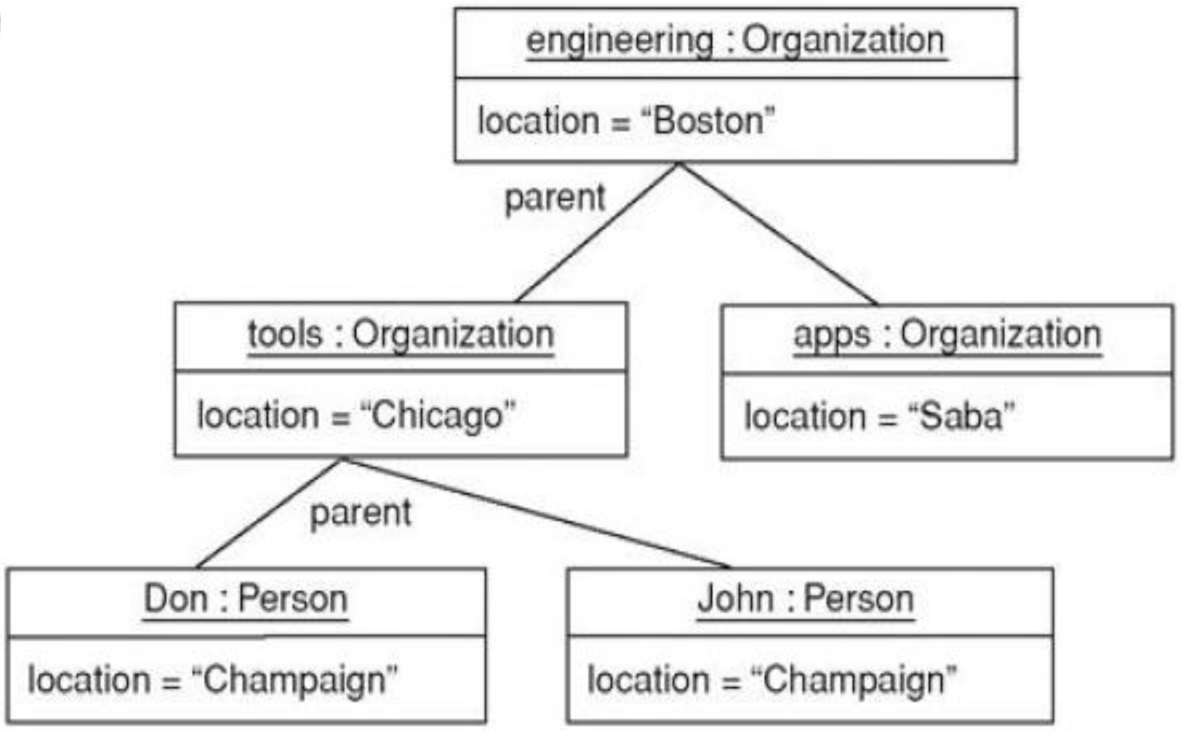
- An **object diagram** is a snapshot of the objects in a system at a point in time.
- Because it shows instances rather than classes, an object diagram is often called an *instance diagram*.
- You can use an object diagram to show an example configuration of objects.

Example



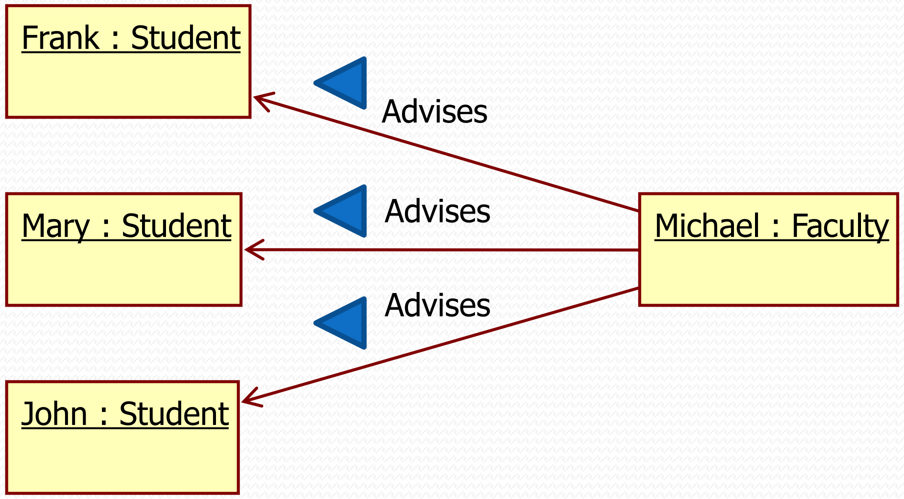
Class Diagram

Object Diagram



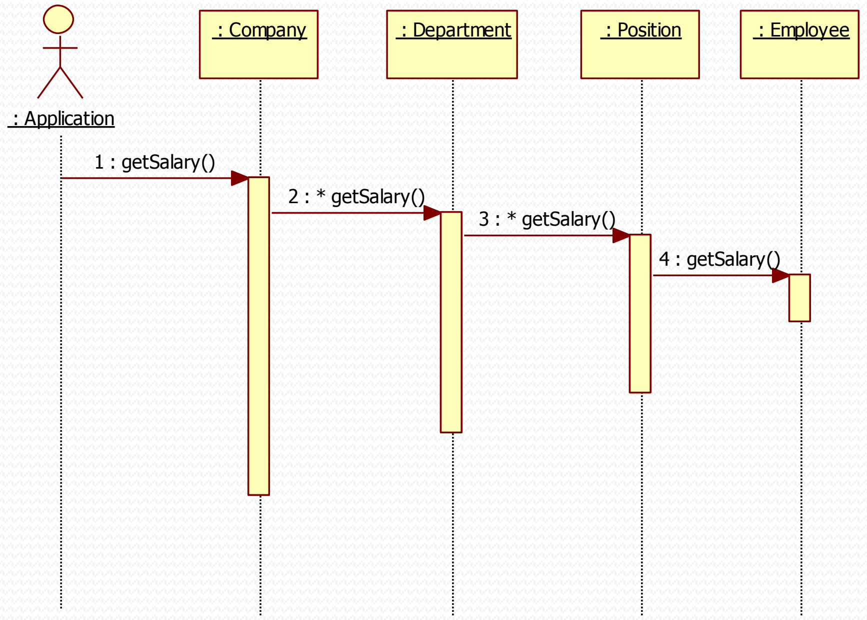
Object Diagram Syntax

- Underlining indicates it's an object
- Usually shows colon separated name and type
- Associations don't have multiplicities
- Associations may or may not display an arrowhead or direction



Exercise

- Create an object diagram that captures a scenario for the computeSalary problem.
- Assume there is one instance of Company, which has two Departments, each with two positions, each filled with one employee. Invent salaries for the employees.



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.

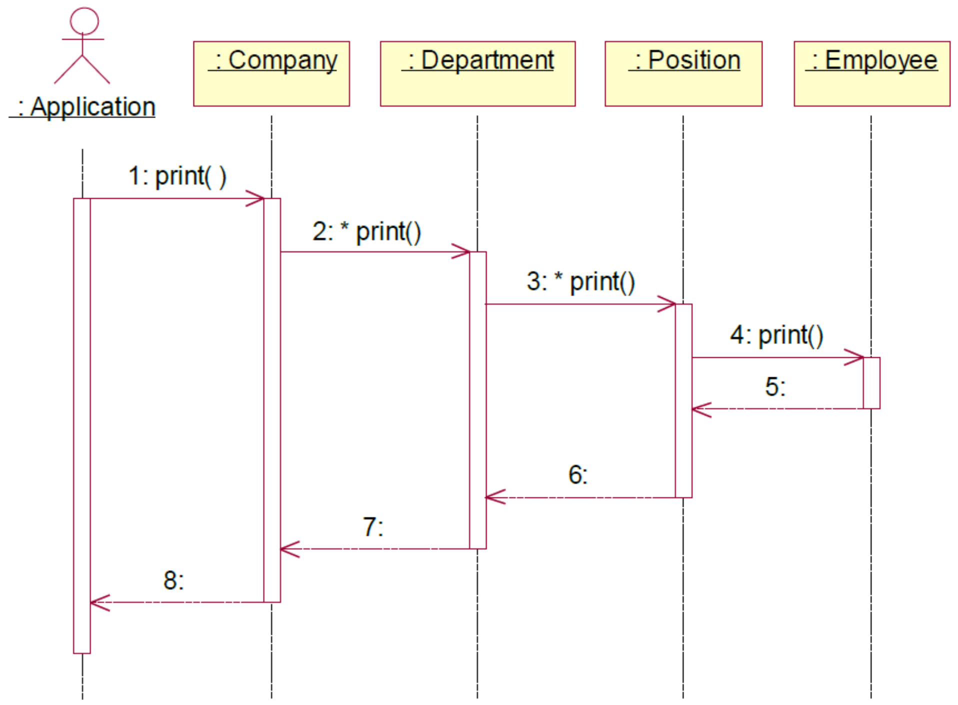
This phenomenon illustrates the principle that *the whole is greater than the sum of the parts*: The objects (parts) are not the important focus for an object diagram. What is important is how the objects relate; together, objects and their relationships form a whole that is more than just the sum of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

Interaction Diagrams: Overview

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

Delegation & Propagation

- A class can express functionality in its interface, but it delegates 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 and propagation.

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

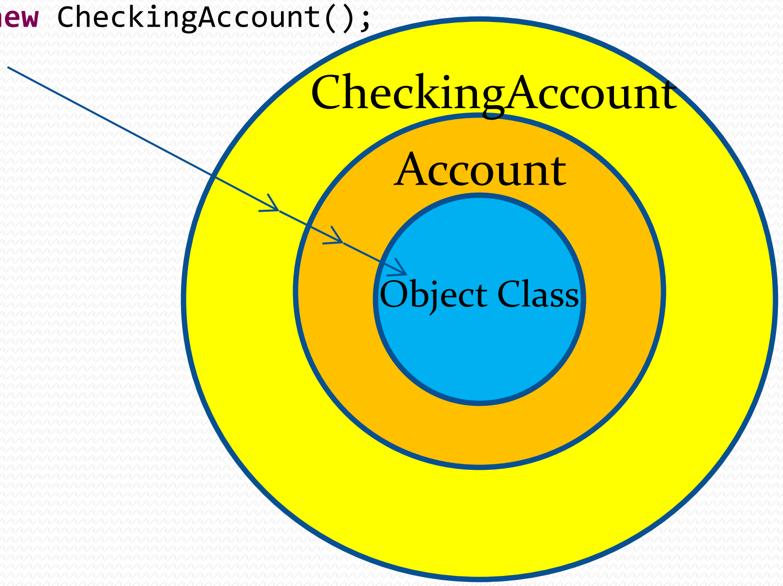
The Open-closed Principle

Separating the change from non-change

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();
```



The method call first checks the class of the actual object to find a `toString()` method if not found, it checks the super Class, up and up until Object

Binding

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

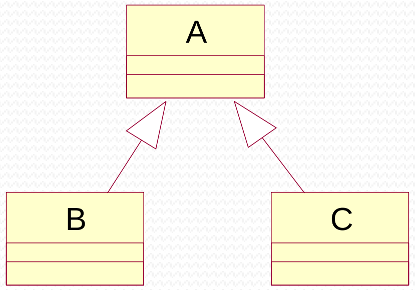
- Static methods have early binding
  - the linkage is made at compile-time.
  - private methods also have early binding (why?)

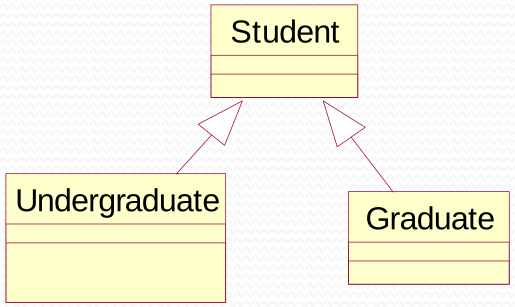
- Late binding, or dynamic binding, occurs at run-time.
  - The JVM method-call mechanism finds the correct method body and invokes it at run-time.
    - by traversing the inheritance chain, starting at the actual class of the object
  - Late binding is what makes polymorphism work

Up-Casting

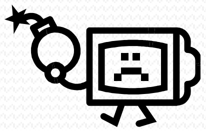
An object of type A can be bound to an object of type B or C

```
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();
```





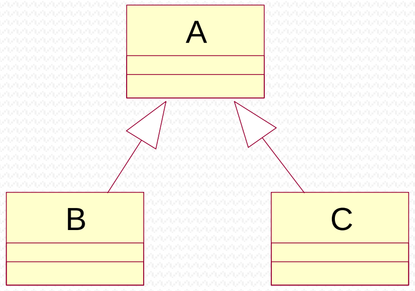
Where is the Compiler Error?

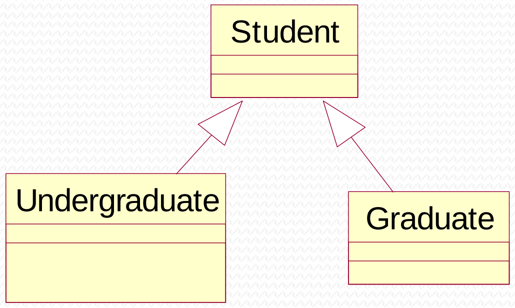


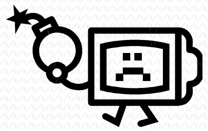
Up-Casting

An object of type A can be bound to an object of type B or C

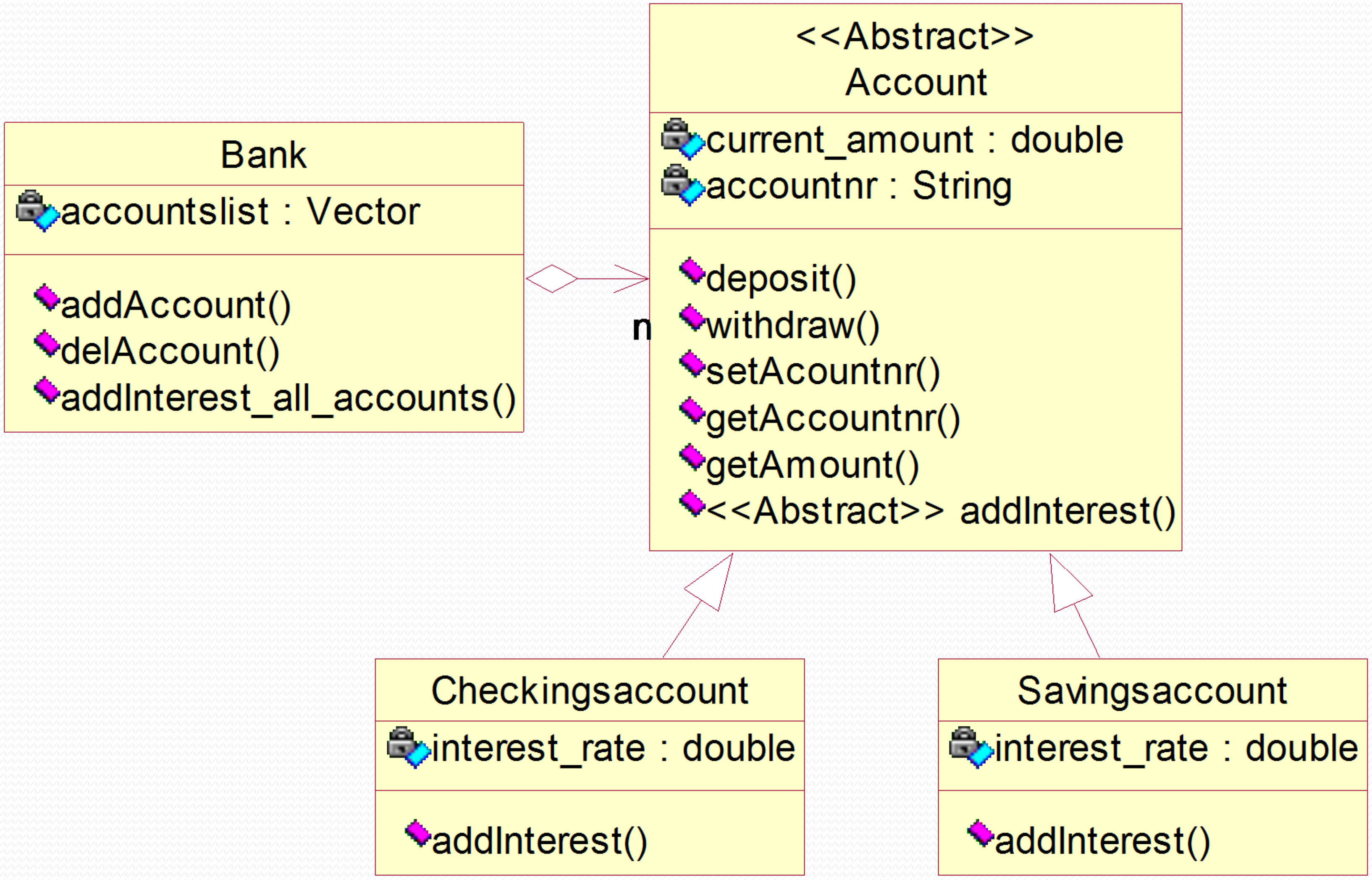
```
public class Student { ... }
public class Undergraduate extends Student { ... }
public class Graduate extends Student { ... }

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







Polymorphism Example



```java
public abstract class Account {
  private double current_amount;
  private String accountnr;

  public void deposit(double amount) {
    current_amount += amount;
  }
  public void withdraw(double amount) {
    current_amount -= amount;
  }

  public void setAccountnr(String anr) {
    accountnr = anr;
  }
  public String getAccountnr() {
    return accountnr;
  }

  public double getAmount() {
    return current_amount;
  }

  public abstract void addInterest();
}
```

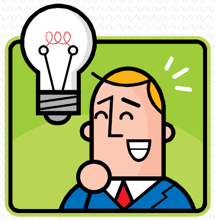
```java
public class CheckingAccount extends Account {
  private double interest_rate = 0.01;

  @Override
  public void addInterest() {
    deposit(getAmount() * interest_rate / 2);
  }
}
```

```java
public class SavingsAccount extends Account {
  private double interest_rate = 0.0425;

  @Override
  public void addInterest() {
    deposit(getAmount() * interest_rate);
  }
}
```

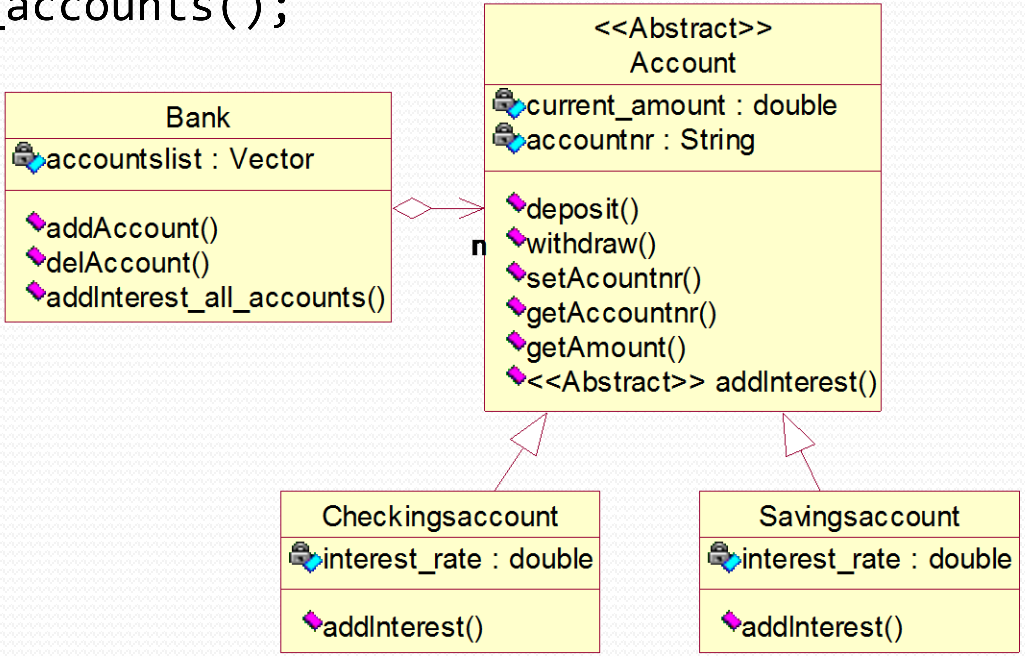
```
public class Bank {
  private Map<String, Account> accounts =
      new HashMap<String, Account>();

  public void addInterest_all_accounts() {
    for (Account a : accounts.values()) {
      a.addInterest();
    }
  }

  public void addAccount(String type, String accountnr) {
    Account account;
    if (type.equals("checking")) {
      account = new CheckingAccount();
    } else {
      account = new SavingsAccount();
    }
    account.setAccountnr(accountnr);
    accounts.put(accountnr, account);
  }

  public void delAccount(String accountnr){
    accounts.remove(accountnr);
  }
}
```

$\leftarrow$ P2I

$\leftarrow$ **Polymorphism**

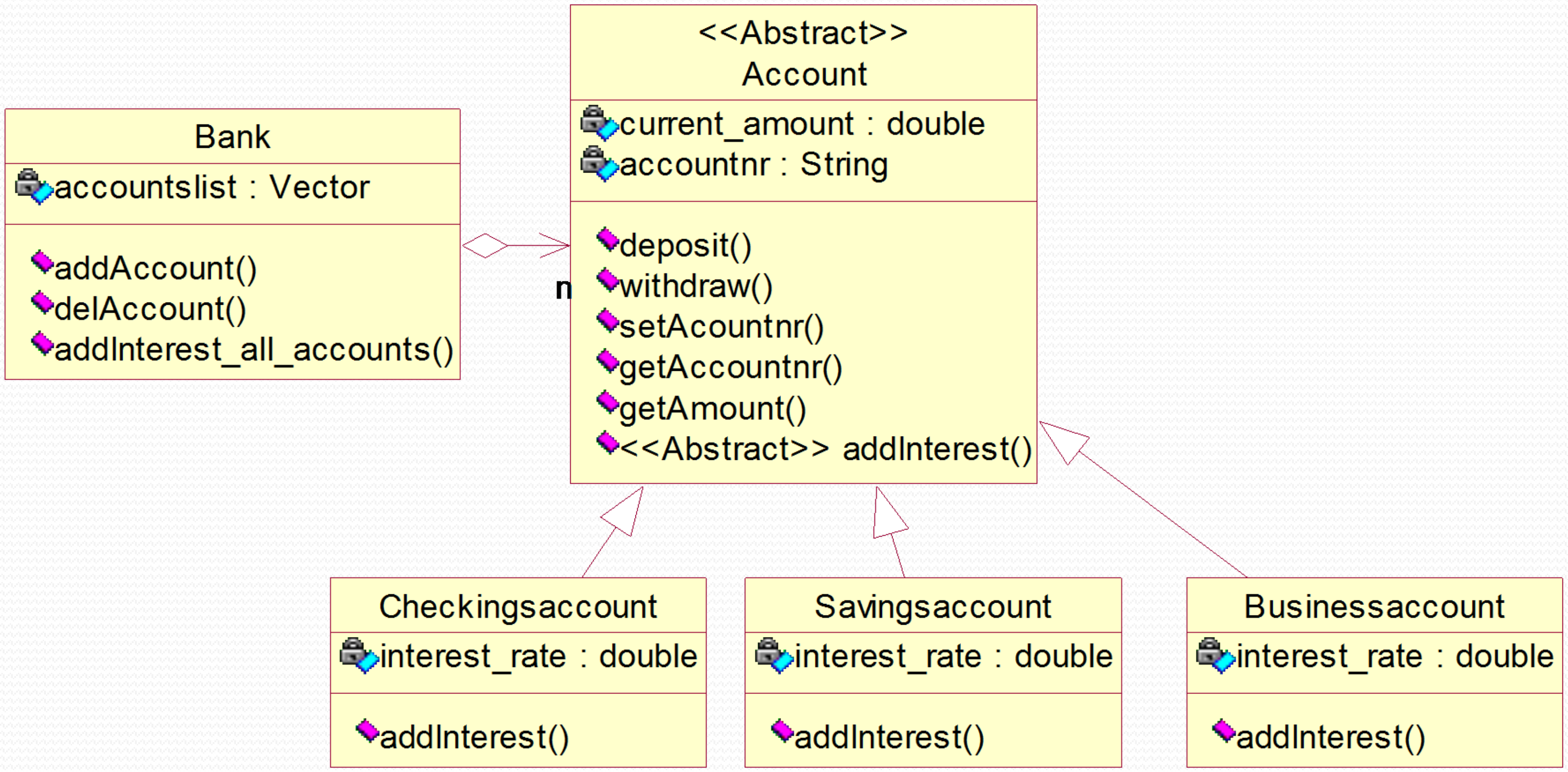


```
public class BankApp {
    public static void main(String[] args) {
        Bank mybank = new Bank();
        mybank.addAccount("checking", "1");
        mybank.addAccount("checking", "2");
        mybank.addAccount("savings",  "3");

        mybank.addInterest_all_accounts();
    }
}
```



Why do we want polymorphism?

- It allows us to **extend** our program with new features without **changing** existing (already tested)code.



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.
- See bank example: new accounts (credit card account, business account) can be added without changing the other classes.
- Software modules can never be 100% closed for modification. Programmer has to decide what aspects should be closed.

Main Point 5

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.

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, propagation, 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. **Transcendental Consciousness** by its very nature, has the fundamental association of self-referral – the Self being aware of the Self
4. **Wholeness moving within itself**: 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