Principles of Software Construction: Objects, Design, and Concurrency

Interaction Diagrams and Introduction to Inheritance

Fall 2014

Charlie Garrod Jonathan Aldrich

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due Thursday
- Homework 2 coming soon!
  - Due next Thursday

Key concepts from last Thursday

The design process

1. Object-oriented analysis
   - Understand the problem
   - Identify the key **concepts** and their relationships
   - Build a (visual) vocabulary
   - Create a **domain model** (aka conceptual model)

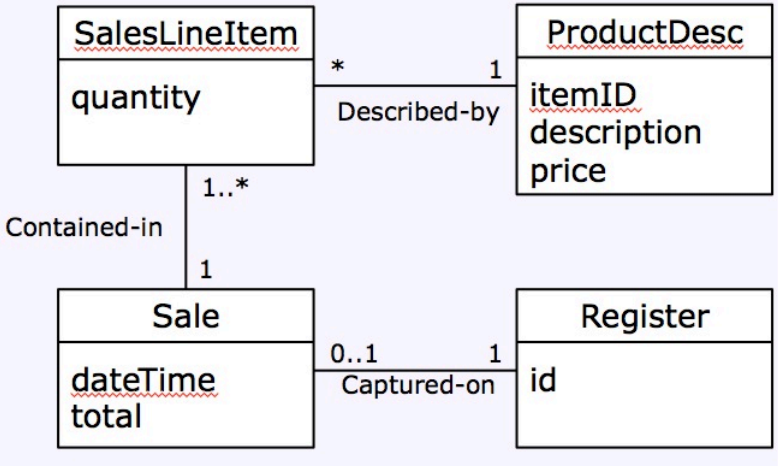
last Thurs

2. Object-oriented design
   - Identify **software classes** and their relationships with *class diagrams*
   - Assign responsibilities (attributes, methods)
   - Explore **behavior** with *interaction diagrams*
   - Explore design alternatives
   - Create an **object model** (aka design model and design class diagram) and **interaction models**

3. Implementation
   - Map designs to code, implementing classes and methods

Documenting a domain model

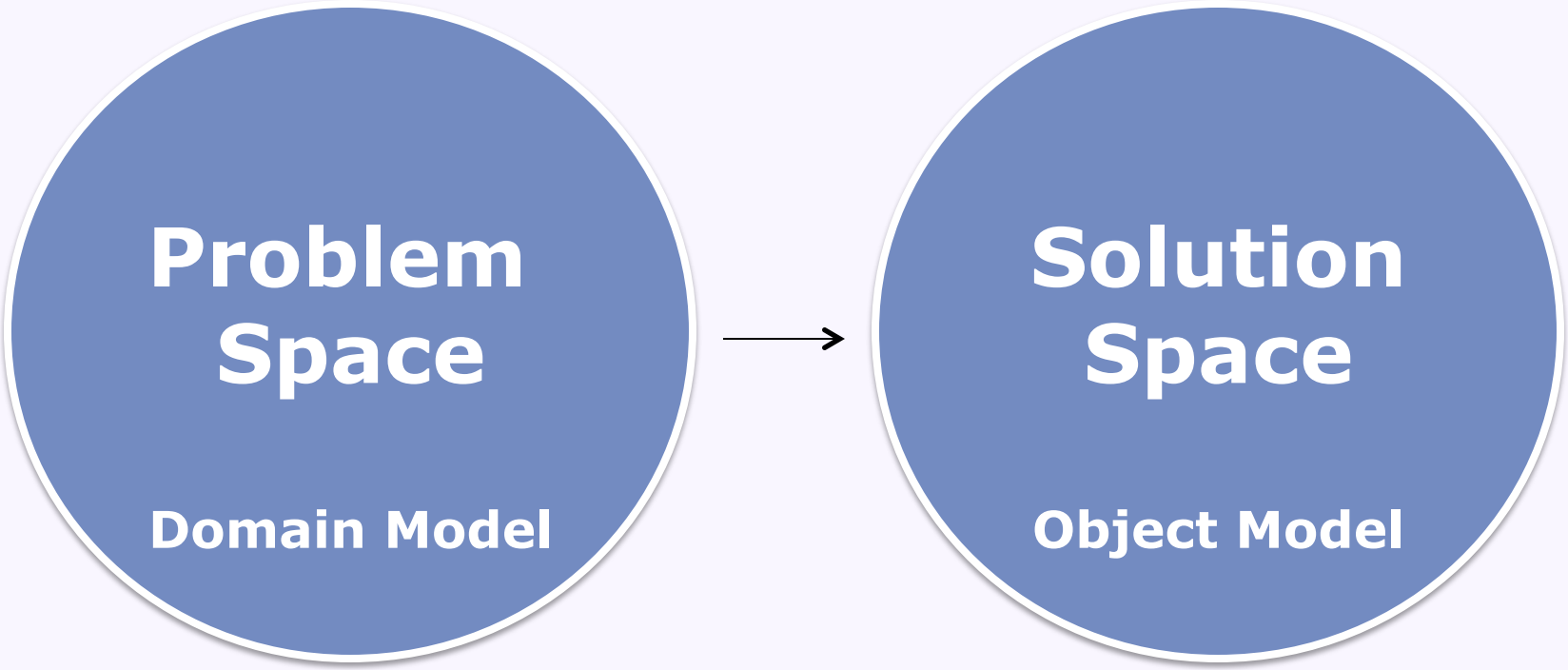
- Typical: UML class diagram
  - Simple classes without methods and essential attributes only
  - Associations, inheritances, etc. as needed
  - Do not include implementation-specific details, e.g., types, method signatures
  - Include notes as needed
- Complement with examples, glossary, etc. as needed
- Formality depends on size of project
- Expect revisions



Today

- Visualizing dynamic behavior
  - Interaction diagrams
- Inheritance and polymorphism
  - For maximal code re-use
  - Design ideas: hierarchical modeling
  - Inheritance and its alternatives
  - Java details related to inheritance

Our design trajectory

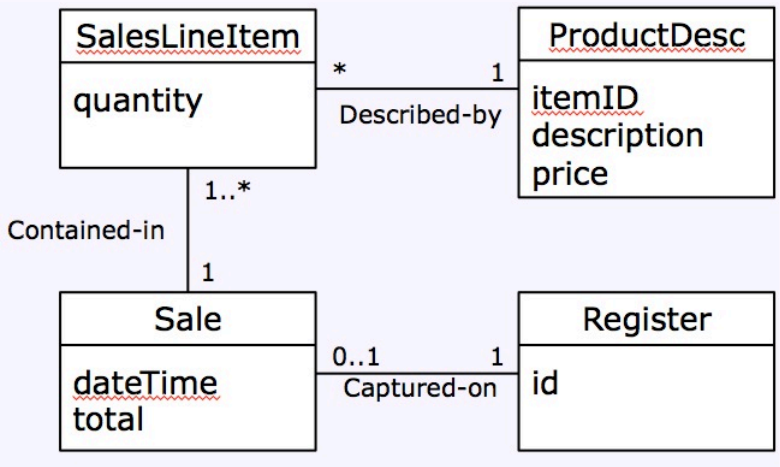


- Real-world concepts
- Requirements, concepts
- Relationships among concepts
- Solving a problem
- Building a vocabulary

- System implementation
- Classes, objects
- References among objects and inheritance hierarchies
- Computing a result
- Finding a solution

How do we bridge the representational gap?

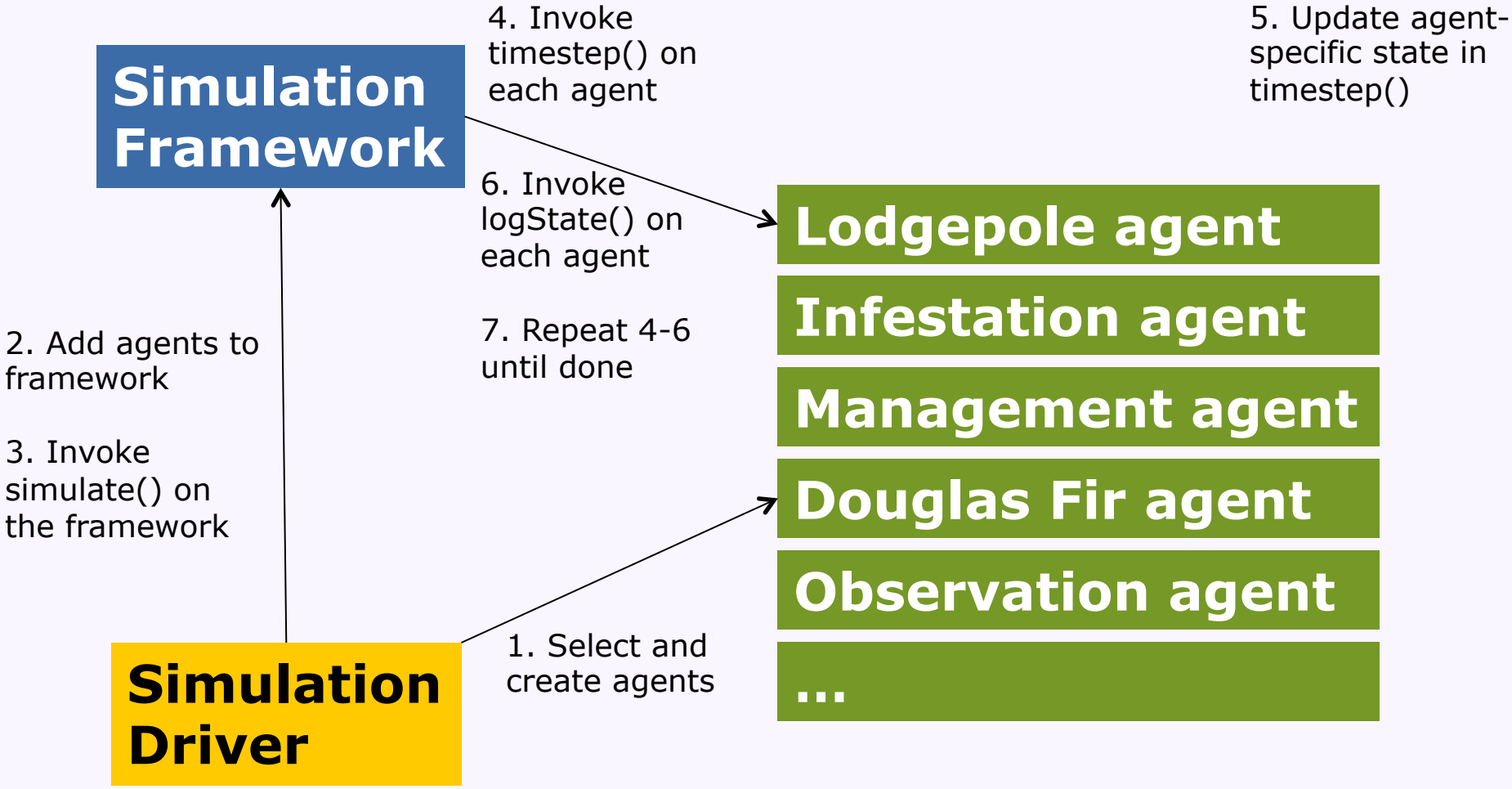
- Domain model with a UML class diagram
  - Simple classes without methods and essential attributes only
  - Associations, inheritances, etc. as needed
  - Do not include implementation-specific details, e.g., types, method signatures
  - Include notes as needed
- Complement with examples, glossary, etc. as needed
- Formality depends on size of project
- Expect revisions



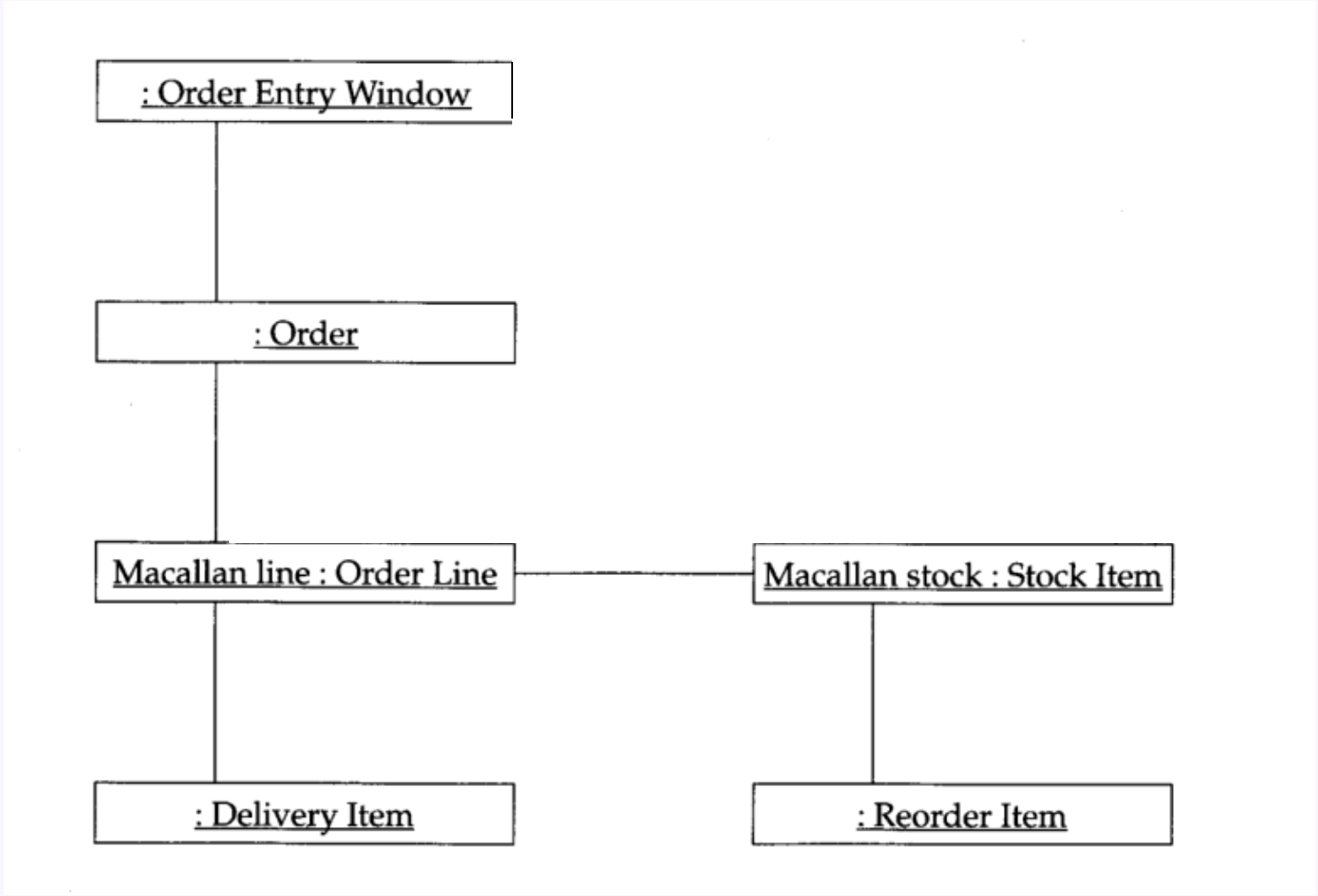
One tool: Interaction diagrams

- An *interaction diagram* is a picture that shows, for a single scenario of use, the events that occur across the system's boundary or between subsystems
- Clarifies interactions:
  - Between the program and its environment
  - Between major parts of the program
- For this course, you should know:
  - Communication diagrams
  - Sequence diagrams

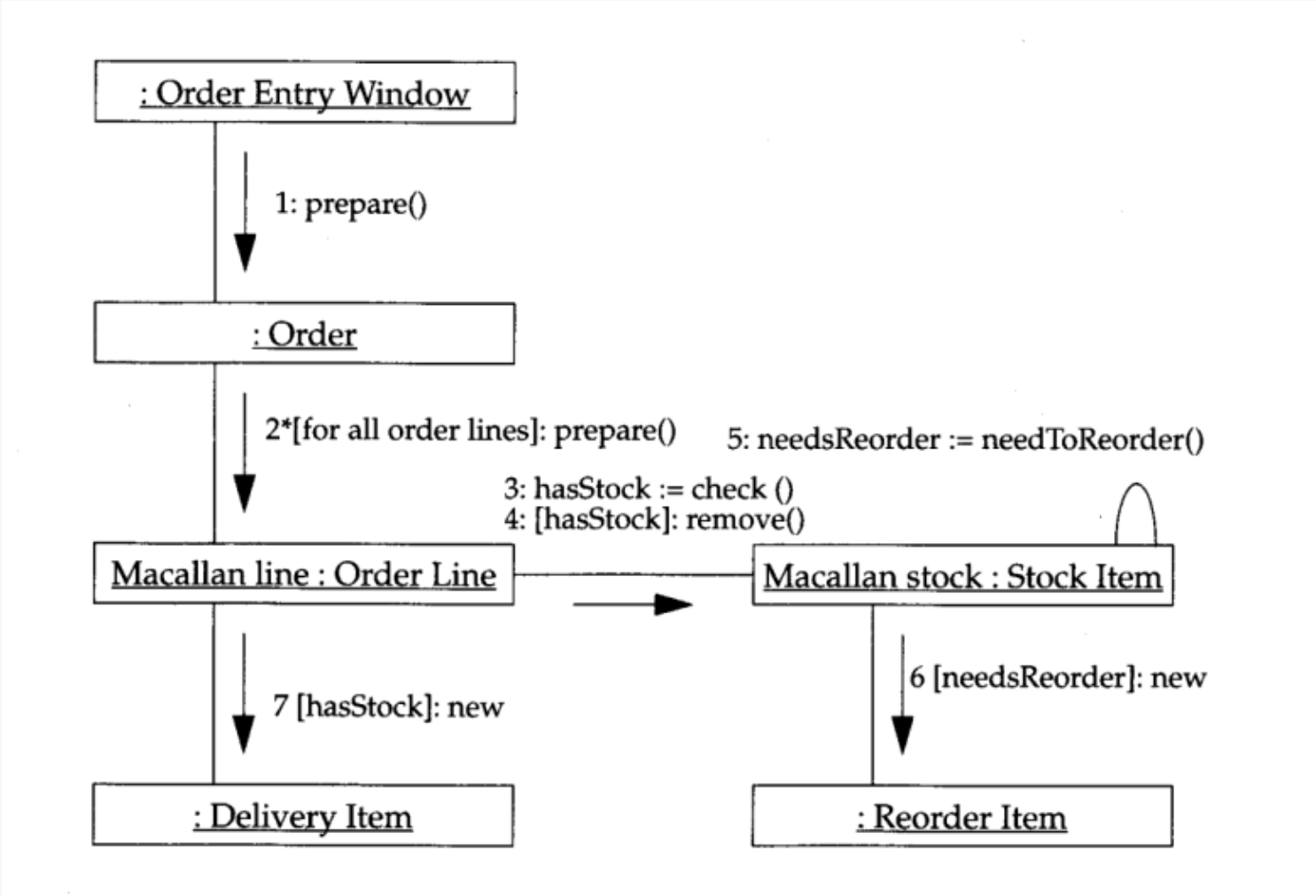
E.g., Lifecycle of Jonathan's simulation framework



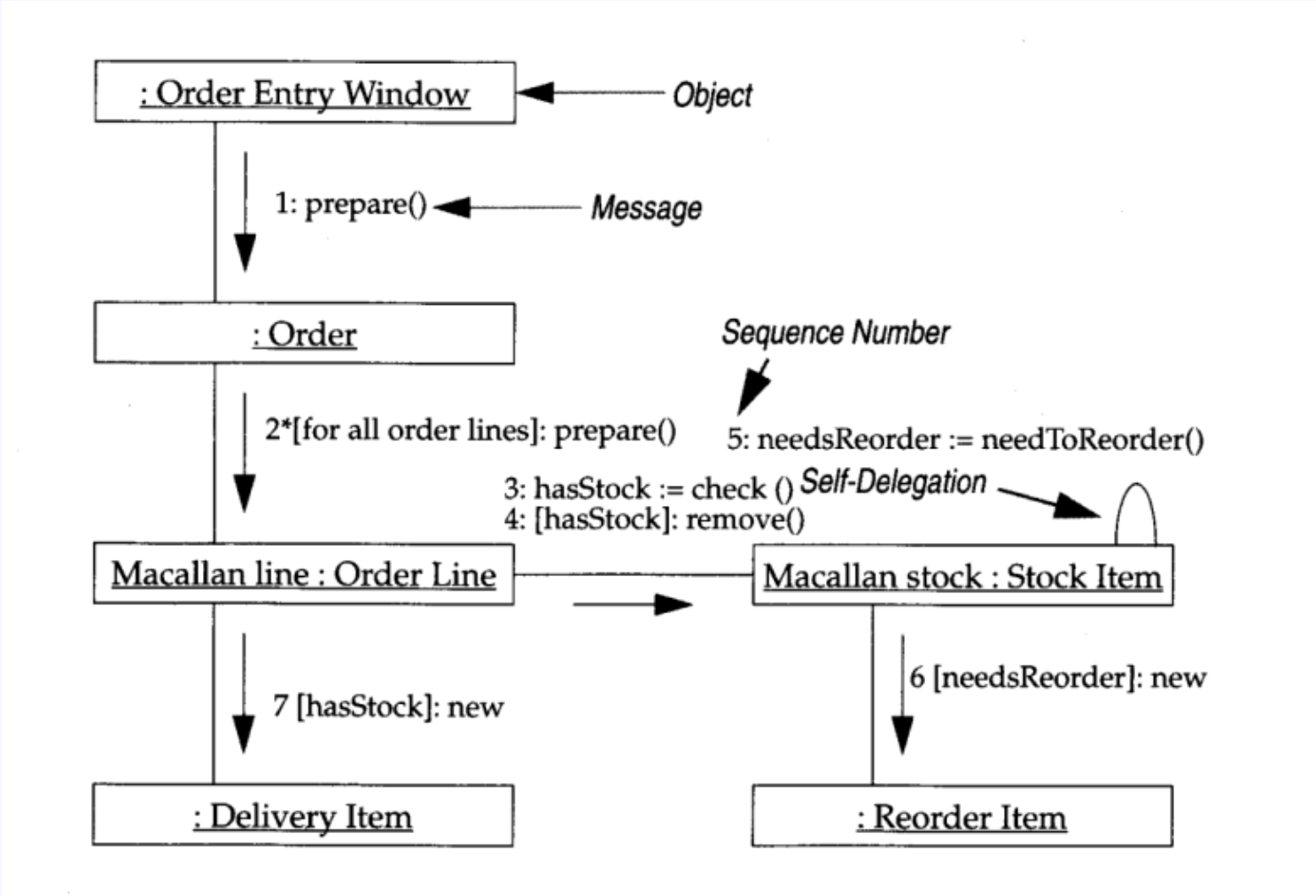
Creating a communication diagram



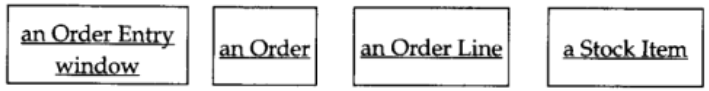
Communication diagram example, complete



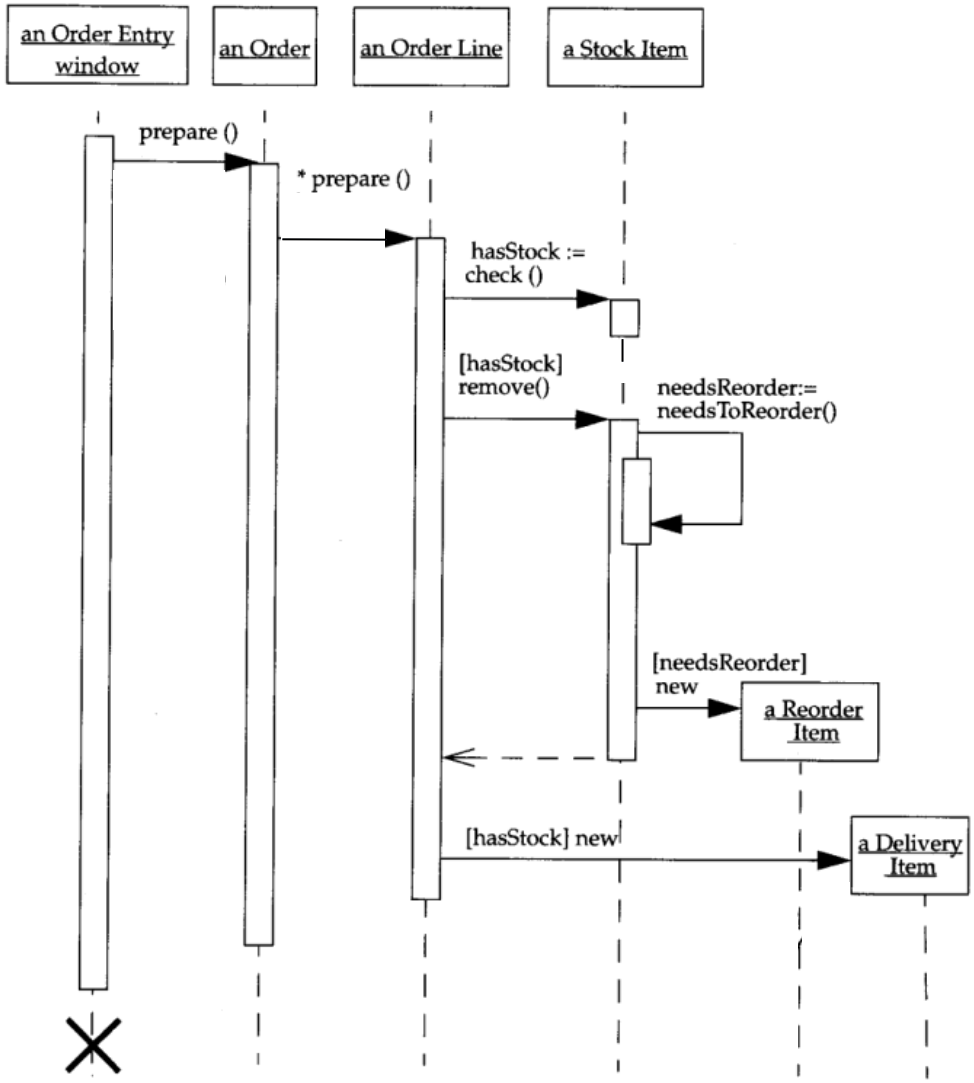
(An annotated communication diagram)



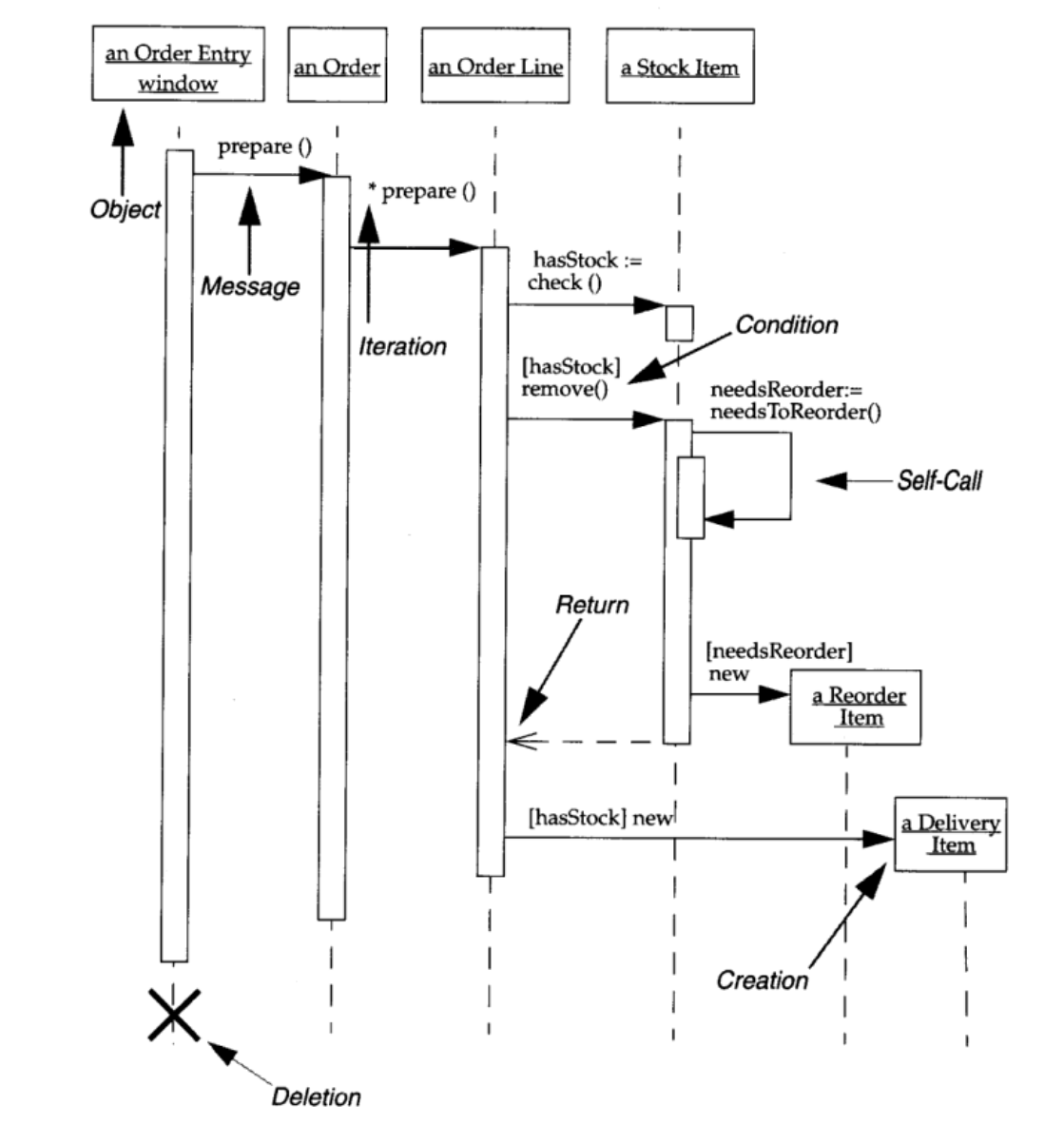
Constructing a sequence diagram



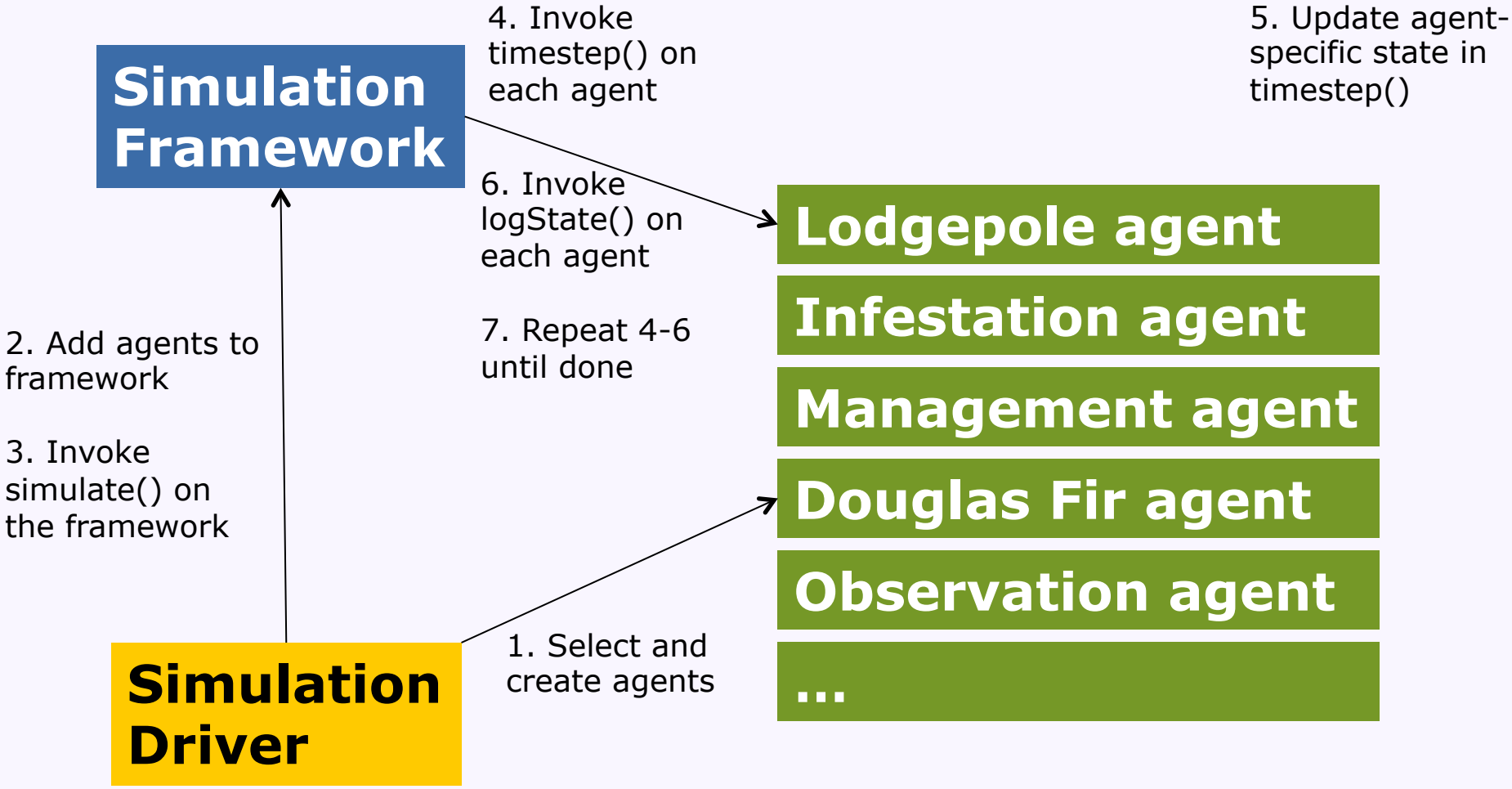
Sequence diagram example, complete



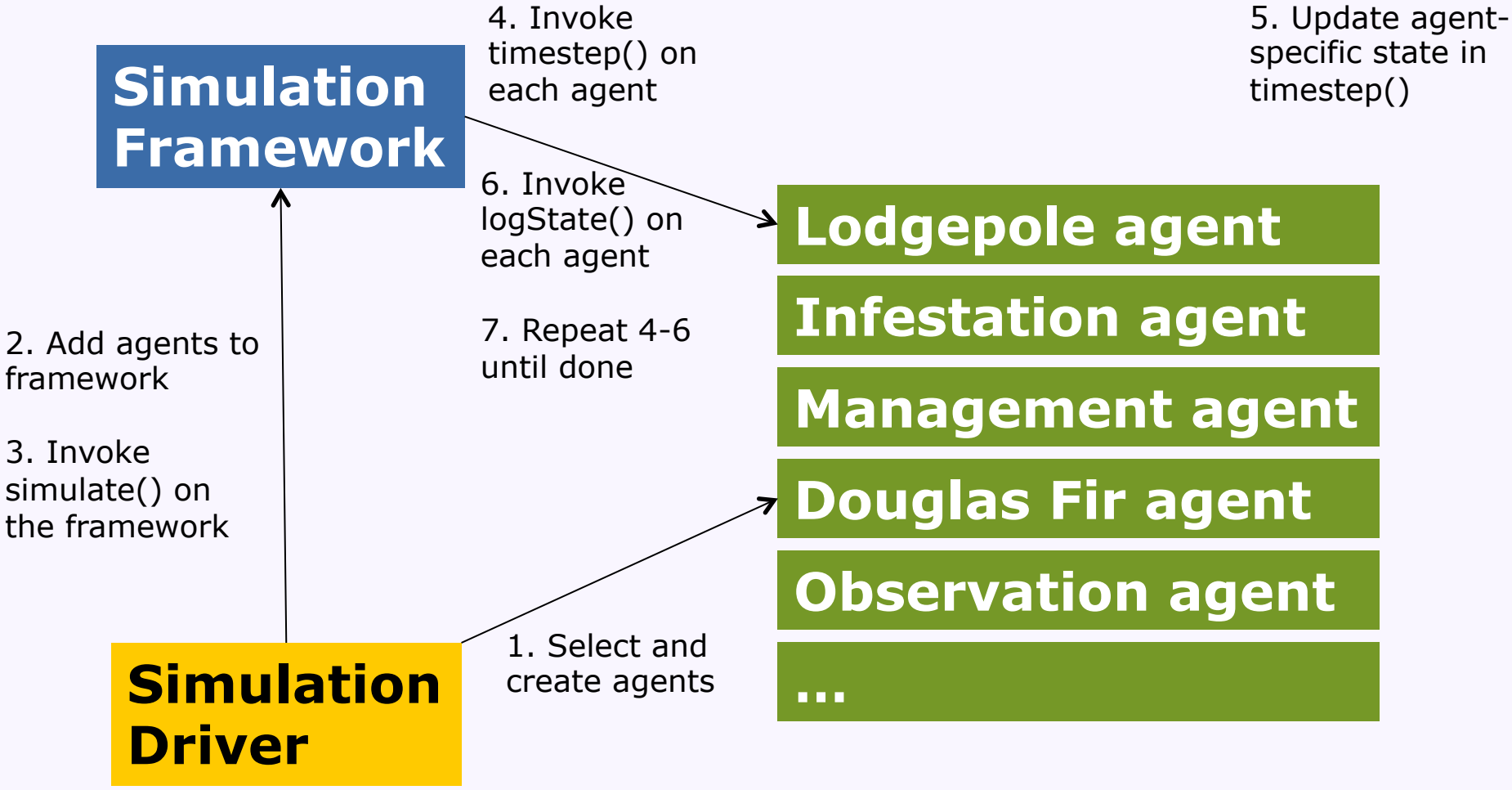
(An annotated sequence diagram)



What kind of interaction diagram is this?



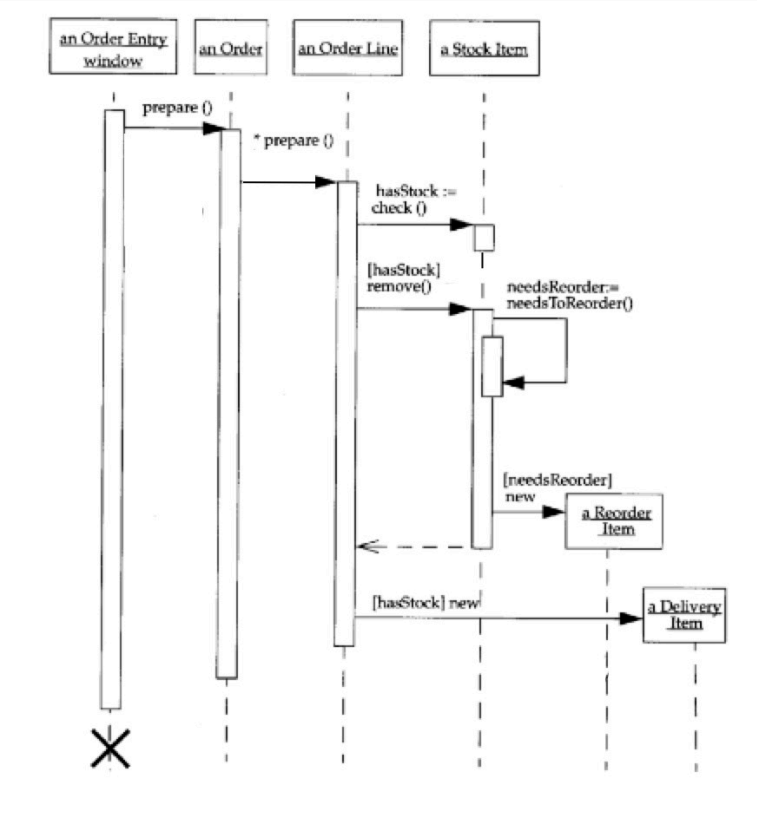
What kind of interaction diagram is this?

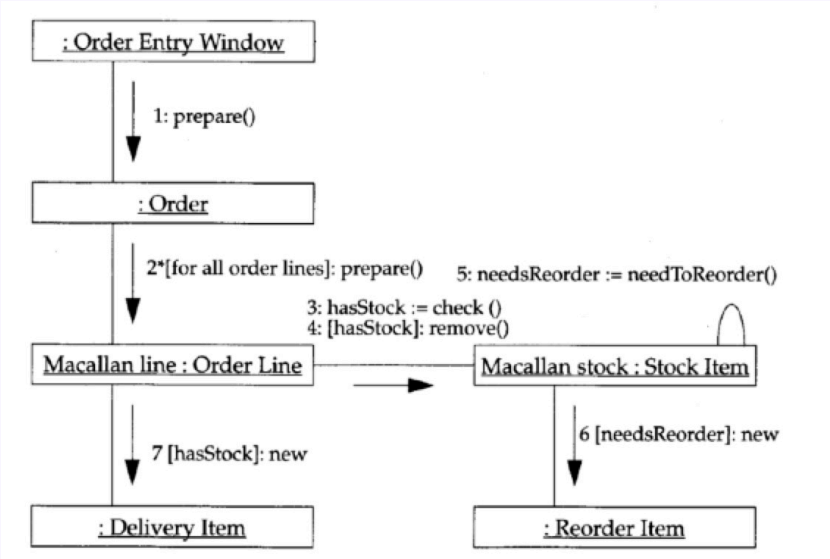


(It is a communication diagram.)

Sequence vs. communication diagrams

- Relative advantages and disadvantages?



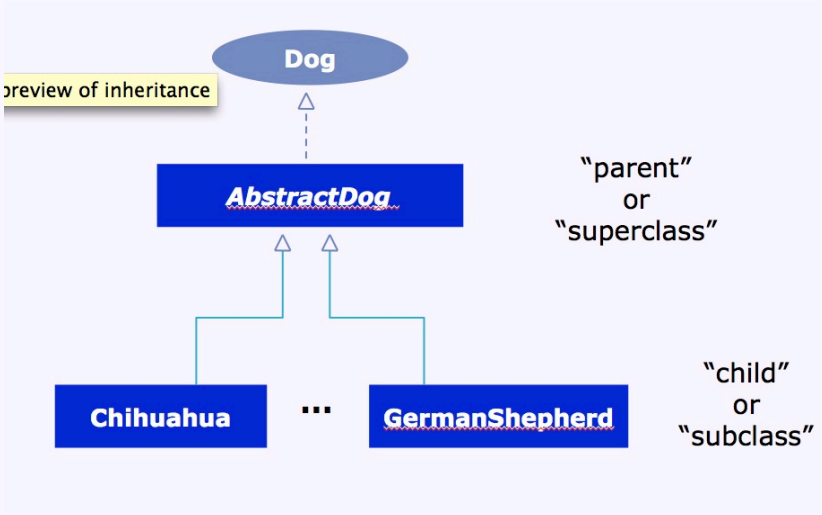


Today

- Visualizing dynamic behavior
  - Interaction diagrams
- Inheritance and polymorphism
  - For maximal code re-use
  - Design ideas: hierarchical modeling
  - Inheritance and its alternatives
  - Java details related to inheritance

An introduction to inheritance

- A dog of an example:
  - Dog.java
  - AbstractDog.java
  - Chihuahua.java
  - GermanShepherd.java

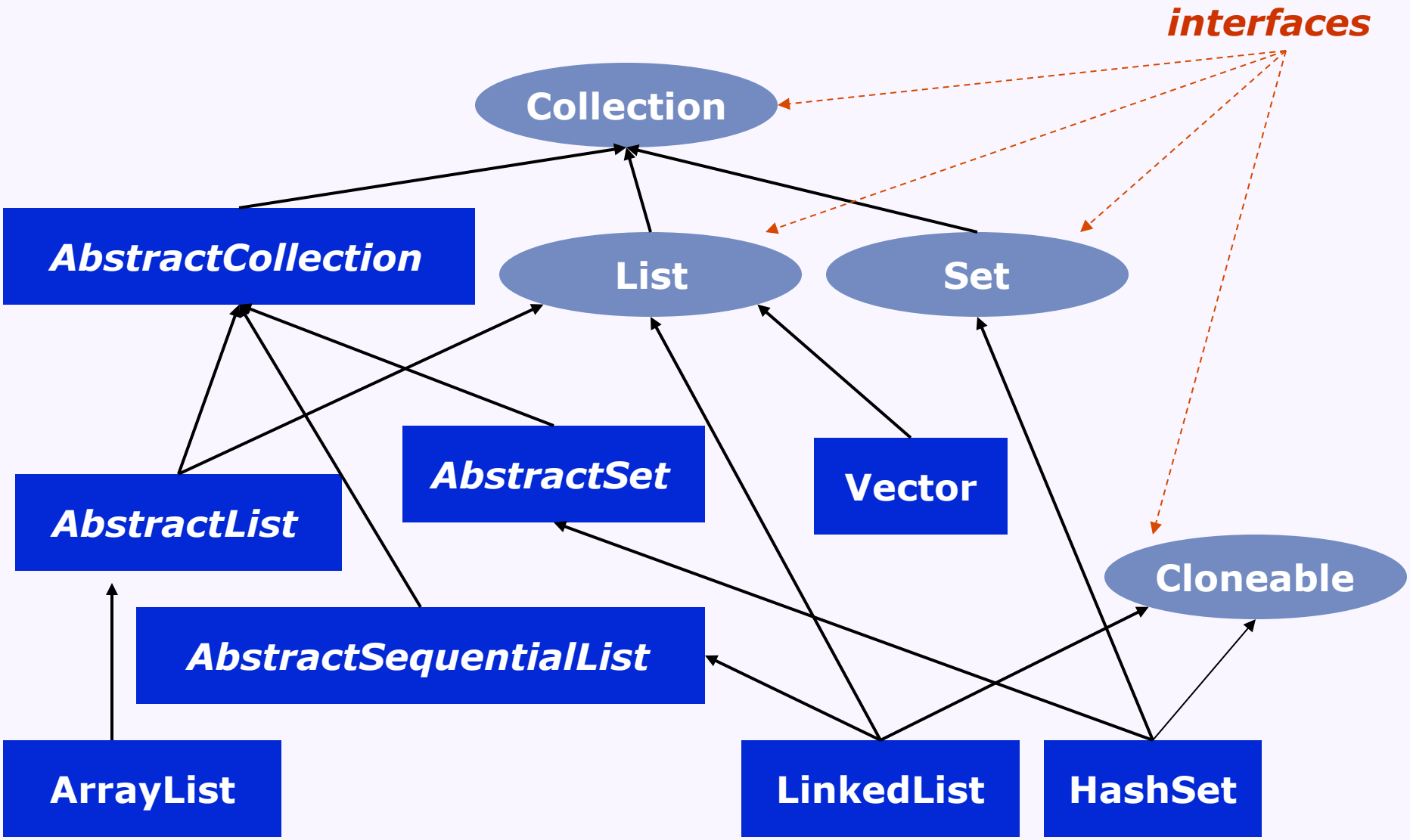


- Typical roles:
  - An interface define expectations / commitment for clients
  - An *abstract class* is a convenient hybrid between an interface and a full implementation
  - Subclass *overrides* a method definition to specialize its implementation

Inheritance: a glimpse at the hierarchy

- Examples from Java
  - `java.lang.Object`
  - Collections library

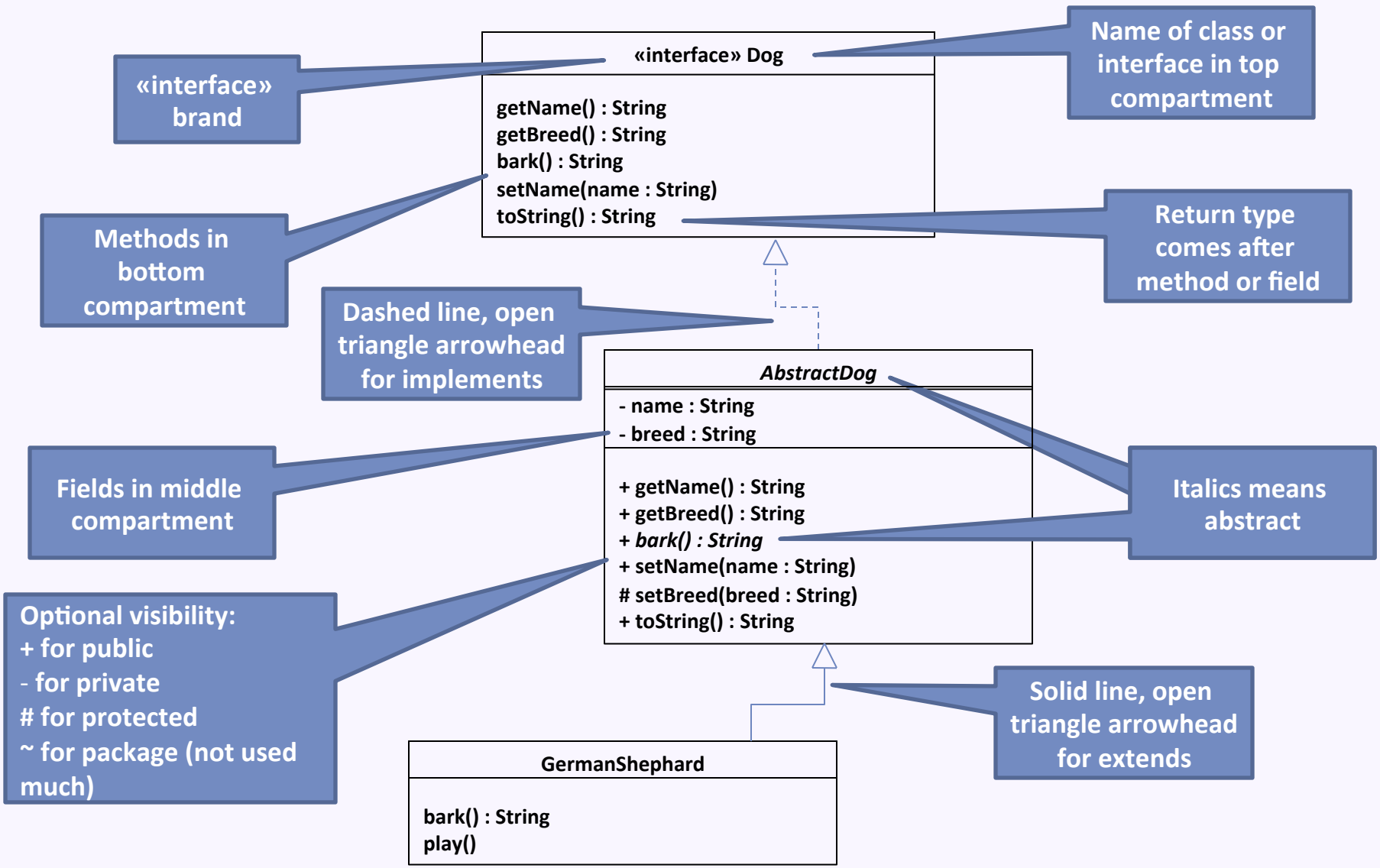
Java Collections API (excerpt)



Benefits of inheritance

- Reuse of code
- Modeling flexibility
- A Java aside:
  - Each class can directly extend only one parent class
  - A class can implement multiple interfaces

Aside: more UML class diagram notation

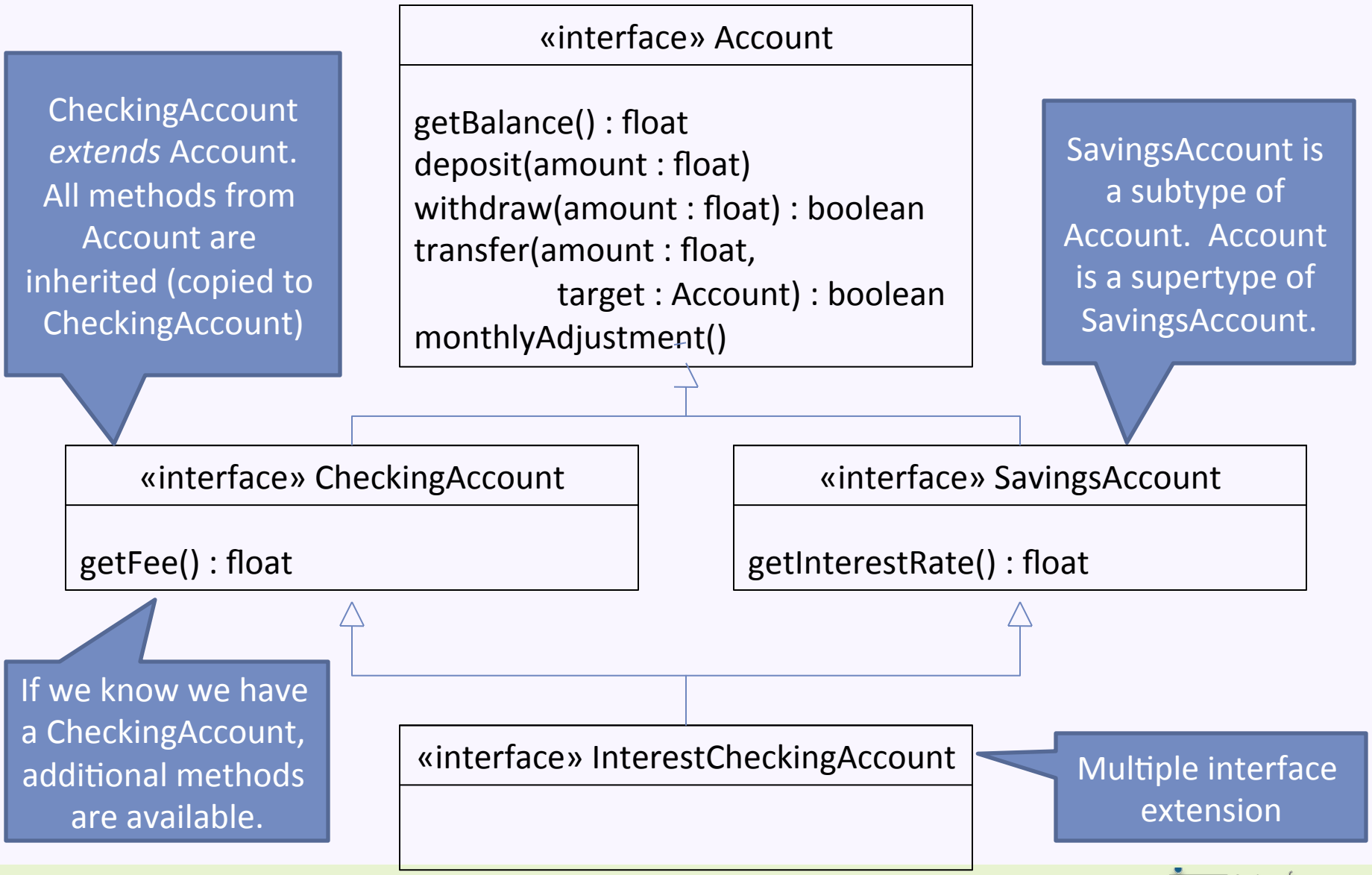


Another example: different kinds of bank accounts

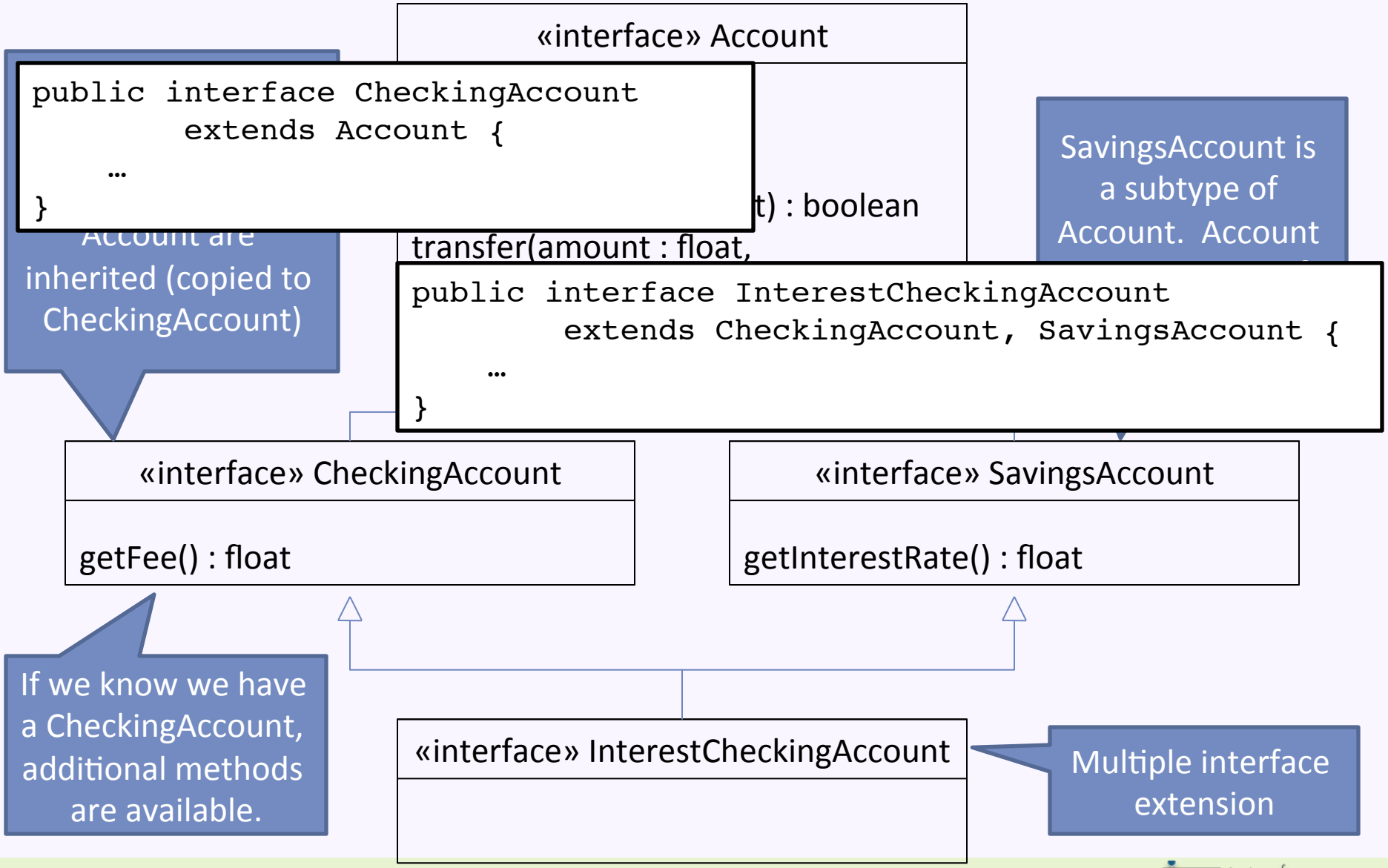
«interface» CheckingAccount
getBalance() : float deposit(amount : float) withdraw(amount : float) : boolean transfer(amount : float, target : Account) : boolean getFee() : float

«interface» SavingsAccount
getBalance() : float deposit(amount : float) withdraw(amount : float) : boolean transfer(amount : float, target : Account) : boolean getInterestRate() : float

A better design: An account type hierarchy



A better design: An account type hierarchy



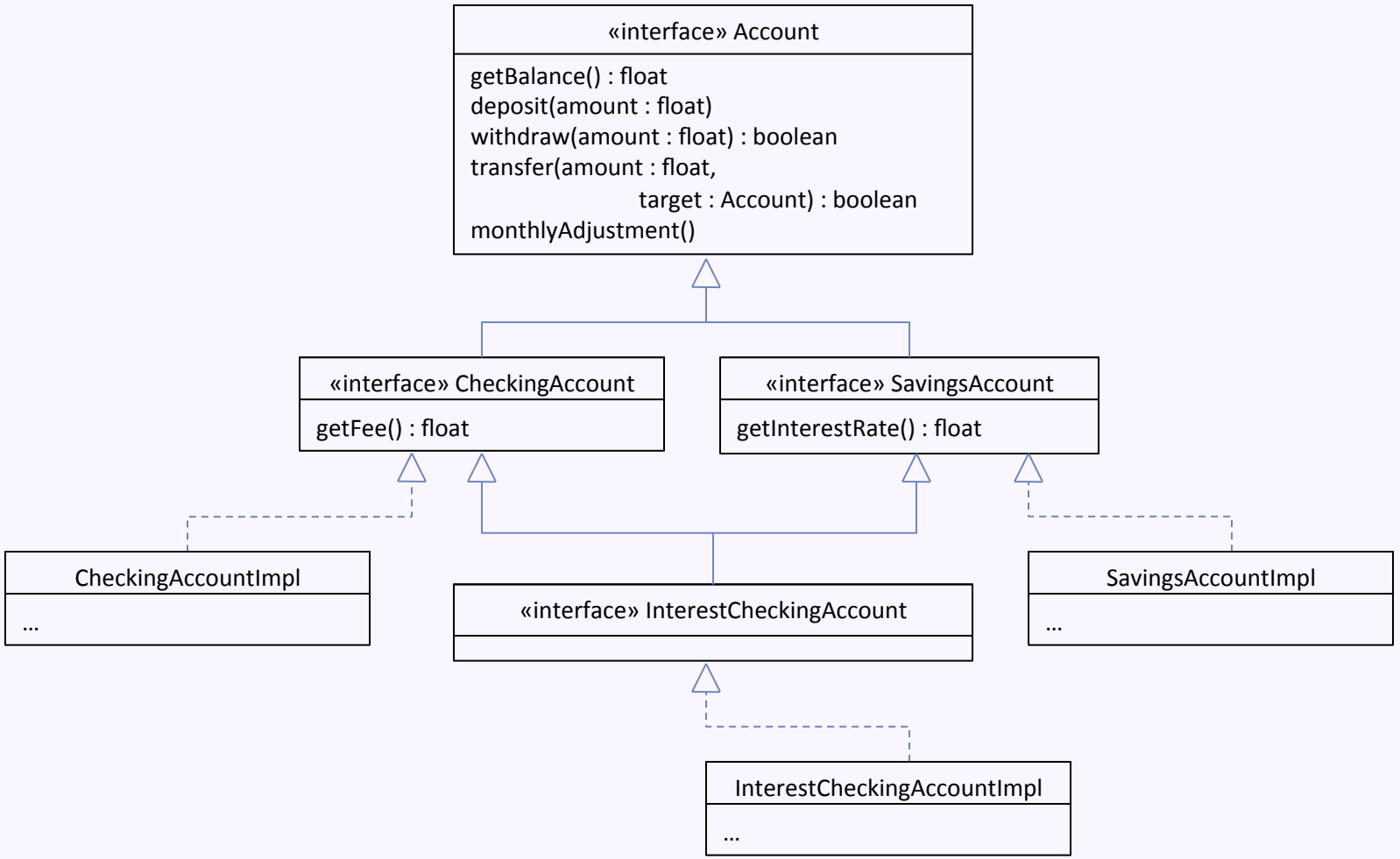
The power of object-oriented interfaces

- Polymorphism
  - Different kinds of objects can be treated uniformly by client code
    - e.g., a list of all accounts
  - Each object behaves according to its type
    - If you add new kind of account, client code does not change
  - Consider this pseudocode:

```
If today is the last day of the month:
   For each acct in allAccounts:
      acct.monthlyAdjustment();
```

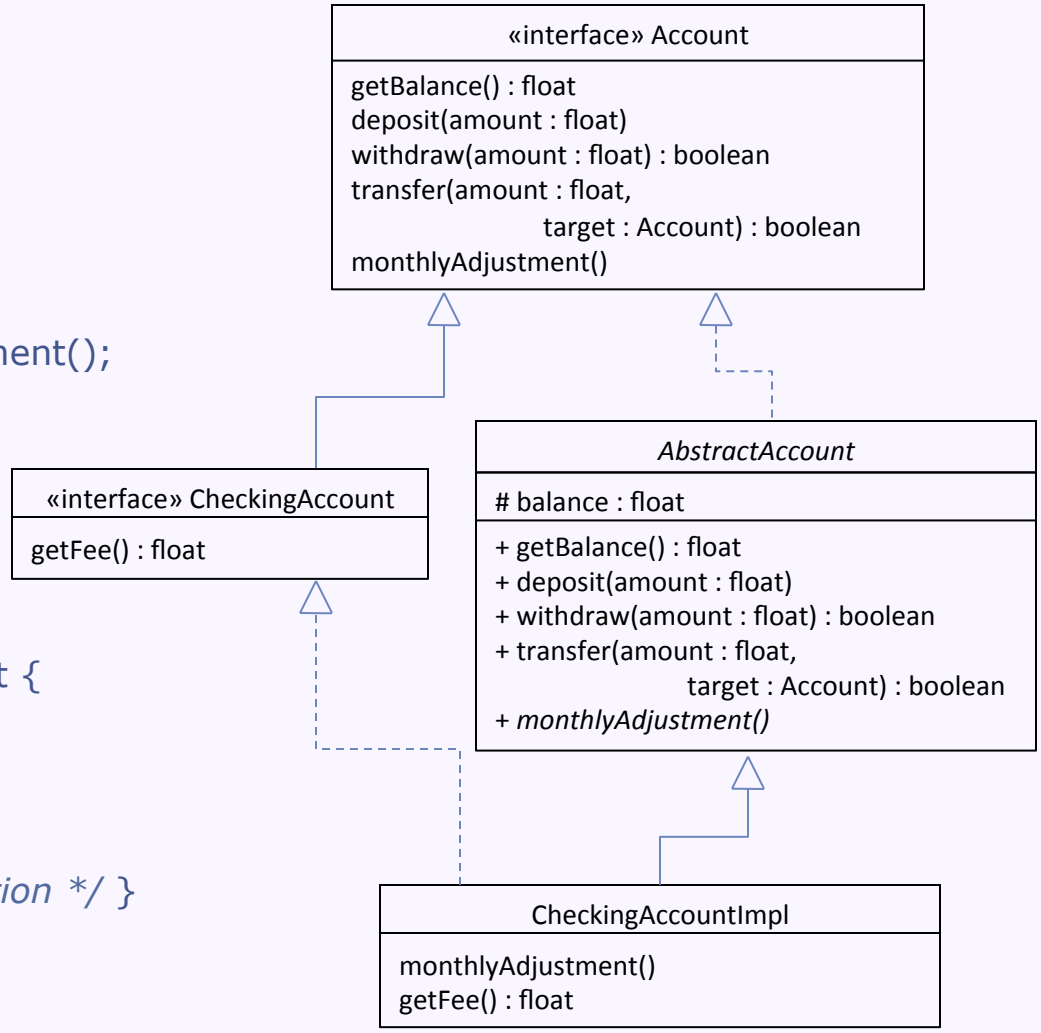
  - See the DogWalker example

One implementation: Just use interface inheritance



Better: Reuse abstract account code

```
public abstract class AbstractAccount
                implements Account {
    protected float balance = 0.0;
    public float getBalance() {
                return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods…
}

public class CheckingAccountImpl
                extends AbstractAcount
                implements CheckingAccount {
    public void monthlyAdjustment() {
                balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```



Better: Reuse abstract account code

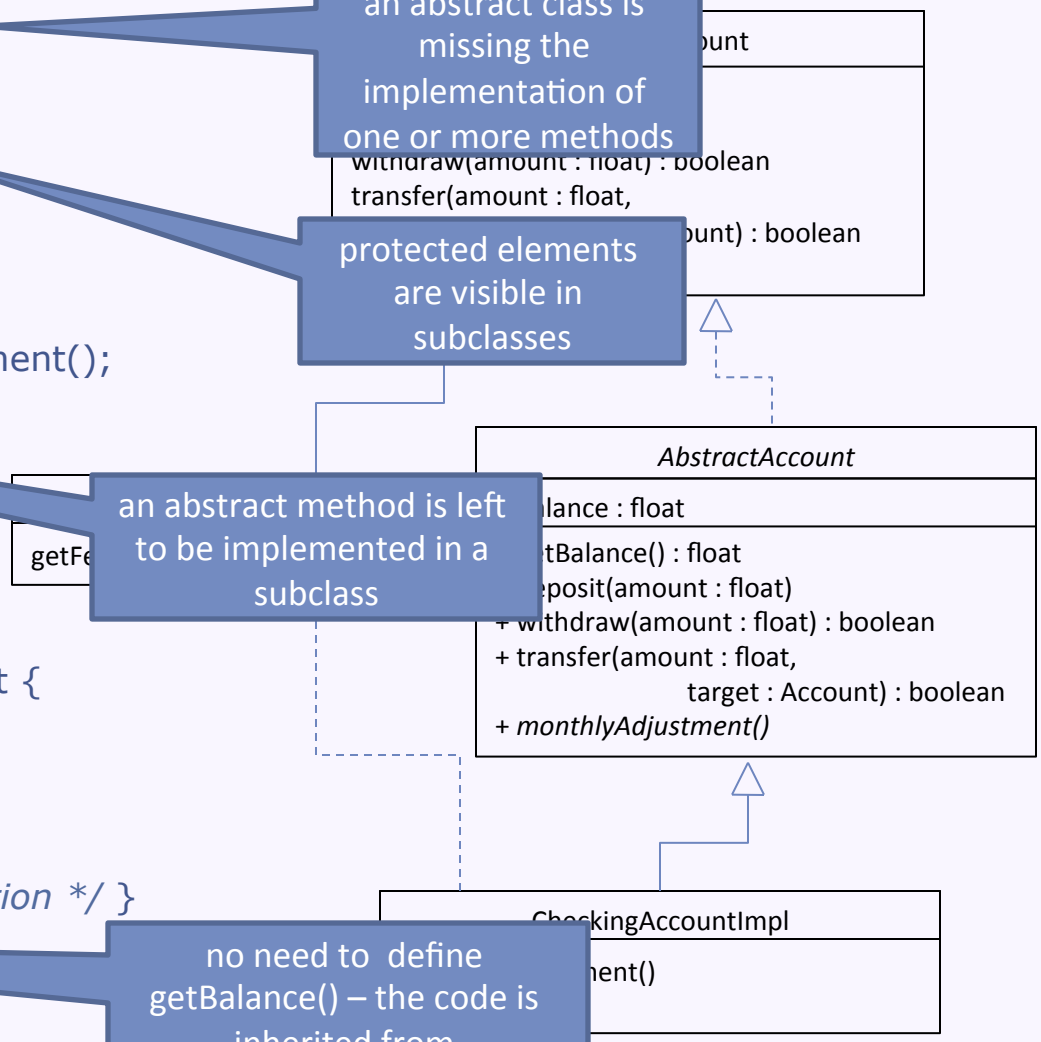
```
public abstract class AbstractAccount
        implements Account {
    protected float balance = 0.0;
    public float getBalance() {
        return balance;
    }
    abstract public void monthlyAdjustment();
    // other methods...
}

public class CheckingAccountImpl
        extends AbstractAcount
        implements CheckingAccount {
    public void monthlyAdjustment() {
        balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```

an abstract class is missing the implementation of one or more methods

protected elements are visible in subclasses

an abstract method is left to be implemented in a subclass

no need to define getBalance() – the code is inherited from AbstractAccount



Inheritance and subtyping

- Inheritance is for code reuse
  - Write code once and only once
  - Superclass features implicitly available in subclass

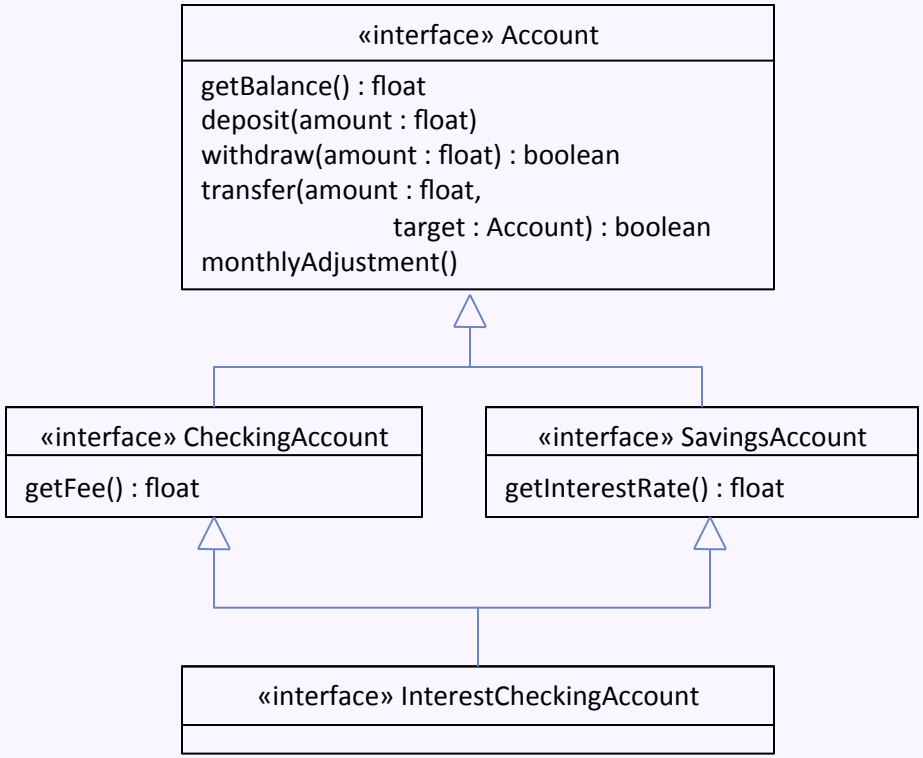
```
class A extends B
```

- Subtyping is for polymorphism
  - Accessing objects the same way, but getting different behavior
  - Subtype is substitutable for supertype

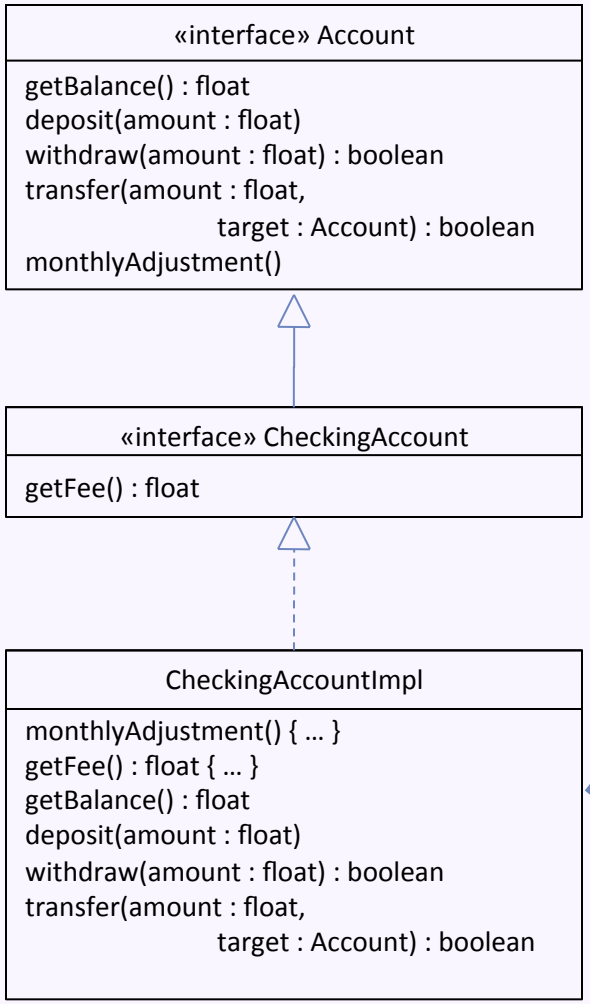
```
class A implements I
class A extends B
```

Challenge: Is inheritance necessary?

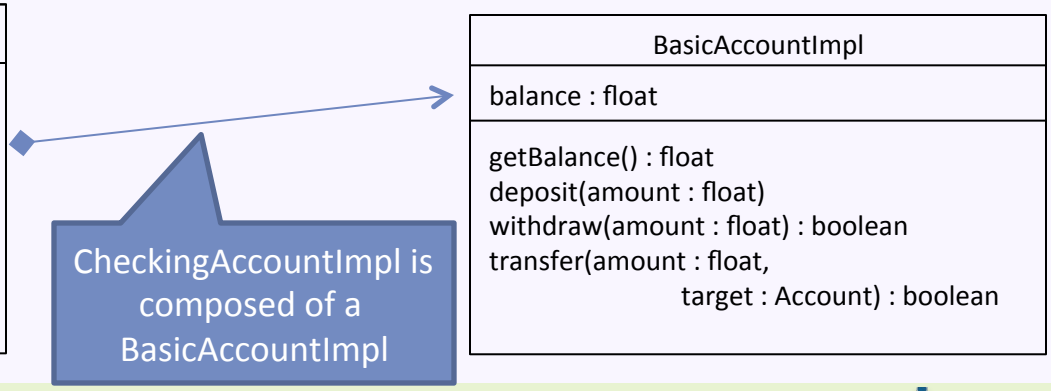
- Can we get the same amount of code reuse without inheritance?



Reuse via *composition* and *delegation*




```
public class CheckingAccountImpl
        implements CheckingAccount {
    BasicAccountImpl basicAcct = new(…);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // …
```



Java details: extended re-use with `super`

```
public abstract class AbstractAccount implements Account {
  protected float balance = 0.0;
  public boolean withdraw(float amount) {
      // withdraws money from account (code not shown)
  }
}

public class ExpensiveCheckingAccountImpl
      extends AbstractAcount implements CheckingAccount {
  public boolean withdraw(float amount) {
      balance -= HUGE_ATM_FEE;
      boolean success = super.withdraw(amount)
      if (!success)
         balance += HUGE_ATM_FEE;
      return success;
  }
}
```

Overrides `withdraw` but also uses the superclass `withdraw` method

Java details: constructors with `this` and `super`

```
public class CheckingAccountImpl
      extends AbstractAcount implements CheckingAccount {

  private float fee;

  public CheckingAccountImpl(float initialBalance, float fee) {
      super(initialBalance);
      this.fee = fee;
  }

  public CheckingAccountImpl(float initialBalance) {
      this(initialBalance, 5.00);
  }
  /* other methods… */ }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

Invokes another constructor in this same class

Java details: `final`

- A final class: prevents extending the class
  - e.g., `public final class CheckingAccountImpl { …`
- A final method: prevents overriding the method
- A final field: prevents assignment to the field
  - (except to initialize it)

- Why might you want to use final in each of the above cases?

Note: type-casting in Java

- Sometimes you want a different type than you have
  - e.g.,
```
float pi = 3.14;
int indianaPi = (int) pi;
```
- Useful if you know you have a more specific subtype:
  - e.g.,
```
Account acct = …;
CheckingAccount checkingAcct =
                        (CheckingAccount) acct;
float fee = checkingAcct.getFee();
```
  - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

Note: `instanceof`

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {
    float adj = 0.0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
        adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
        adj = savingsAcct.getInterest();
    }
    …
}
```

- Advice: avoid `instanceof` if possible
  - Never(?) use `instanceof` in superclass to check type against subclass

Avoiding `instanceof` with the Template Method pattern

```
public interface Account {
    …
    public float getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    …
    public float getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    …
    public float getMonthlyAdjustment() {
        return getInterest();
    }
}
```

Avoiding `instanceof` with the Template Method pattern

```
float adj = 0.0;
if (acct instanceof CheckingAccount) {
    checkingAcct = (CheckingAccount) acct;
    adj = checkingAcct.getFee();
} else if (acct instanceof SavingsAccount) {
    savingsAcct = (SavingsAccount) acct;
    adj = savingsAcct.getInterest();
}
```

Instead:

```
float adj = acct.getMonthlyAdjustment();
```

Thursday…

- Behavioral contracts and subtyping rules