

LECTURE

11

OBJECT-ORIENTED DESIGN: INTRODUCTION

Lecture Goals

- ❑ To learn how to discover new classes and methods
- ❑ To use CRC cards for class discovery
- ❑ To understand the concepts of cohesion and coupling
- ❑ To identify inheritance, aggregation, and dependency relationships between classes
- ❑ To describe class relationships using UML class diagrams
- ❑ To apply object-oriented design techniques to building complex programs
- ❑ To use packages to organize programs

Contents

- ❑ Classes and Their Responsibilities
- ❑ Relationships Between Classes
- ❑ Application: Printing an Invoice

12.1 Classes and Their Responsibilities (1)

- ❑ To discover classes, look for nouns in the problem description
 - Example: Print an invoice
 - Candidate classes:
 - Invoice
 - LineItem
 - Customer

INVOICE

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Item	Qty	Price	Total
Toaster	3	\$29.95	\$89.85
Hair Dryer	1	\$24.95	\$24.95
Car Vacuum	2	\$19.99	\$39.98

AMOUNT DUE: \$ 154.78

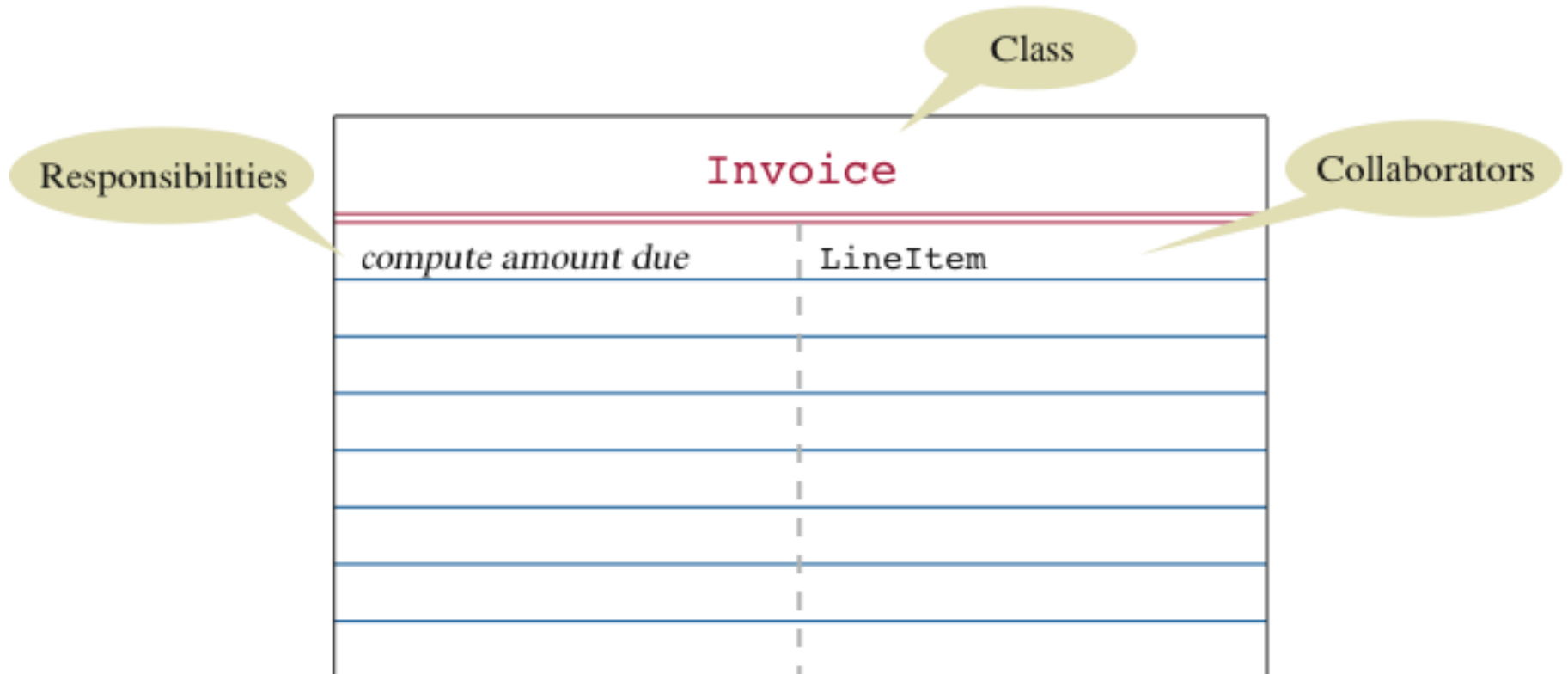
12.1 Classes and Their Responsibilities (2)

- ❑ Concepts from the problem domain are good candidates for classes
 - Examples:
 - From science: Cannonball
 - From business: CashRegister
 - From a game: Monster
- ❑ The name for such a class should be a noun that describes the class

The CRC Card Method

- A **CRC card** describes a class, its responsibilities, and its collaborating classes.
 - For each responsibility of a class, its **collaborators** are the other classes needed to fulfill it

CRC Card



Cohesion (1)

- ❑ A class should represent a single concept
- ❑ The public interface of a class is **cohesive** if all of its features are related to the concept that the class represents

Cohesion (2)

❑ This class lacks cohesion:

```
public class CashRegister
{
    public static final double NICKEL_VALUE = 0.05;
    public static final double DIME_VALUE = 0.1;
    public static final double QUARTER_VALUE = 0.25;
    ...
    public void enterPayment(int dollars, int quarters,
                             int dimes, int nickels, int pennies)
    . . .
}
```

❑ It involves two concepts: *cash register* and *coin*

Cohesion (3)

❑ Better: Make two classes:

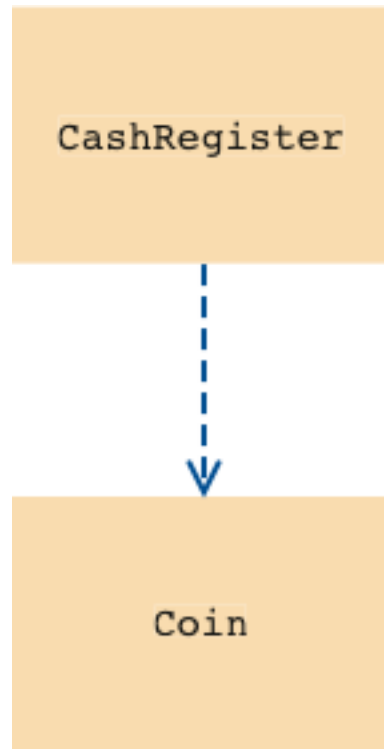
```
public class Coin
{
    public Coin(double aValue, String aName) { . . . }
    public double getValue() { . . . }
    . . .
}

public class CashRegister
{
    public void enterPayment(int coinCount, Coin coinType)
    { . . . }
    . . .
}
```

12.2 Relationships Between Classes

- ❑ A class **depends** on another if it uses objects of that class
 - “knows about” relationship
- ❑ `CashRegister` depends on `Coin` to determine the value of the payment
- ❑ **UML**: to visualize relationships, draw class diagrams

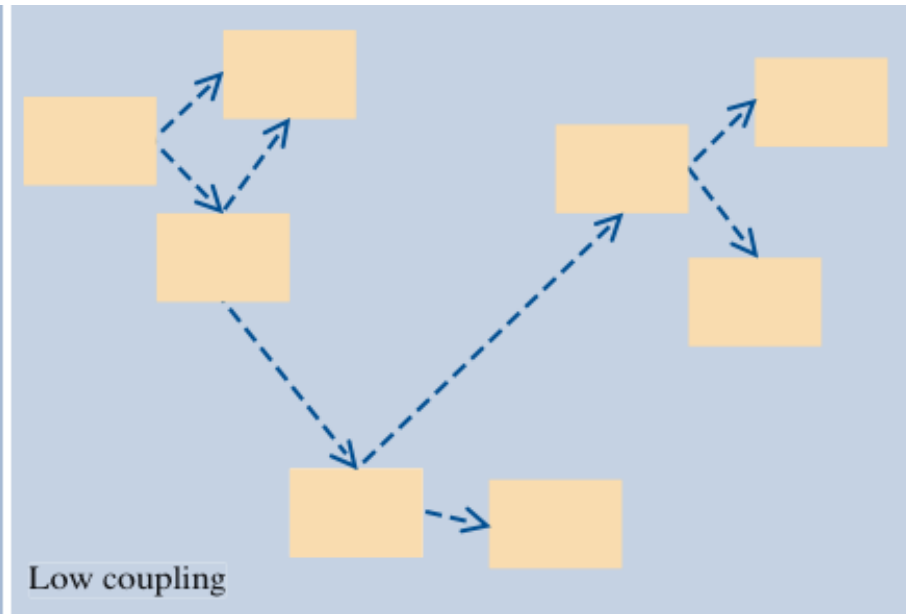
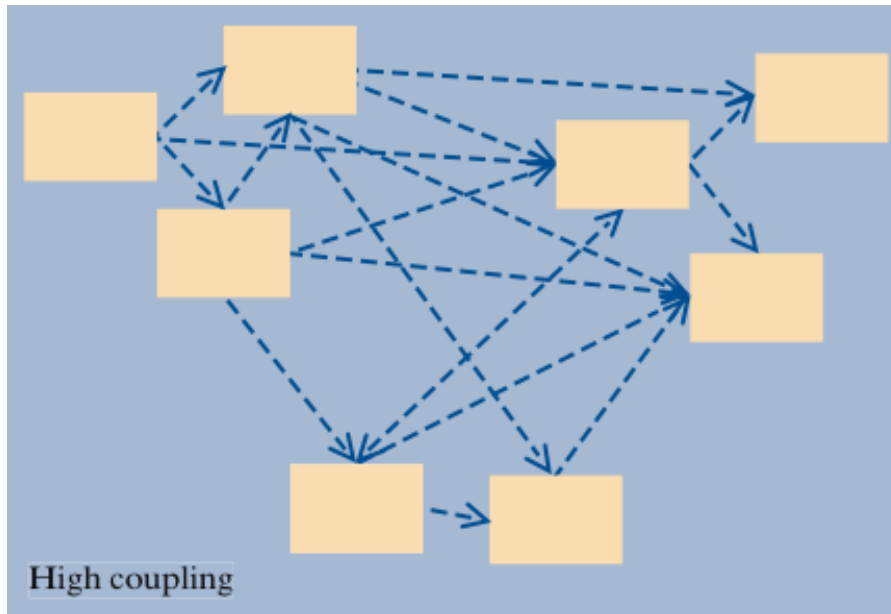
Dependency Relationship



Coupling (1)

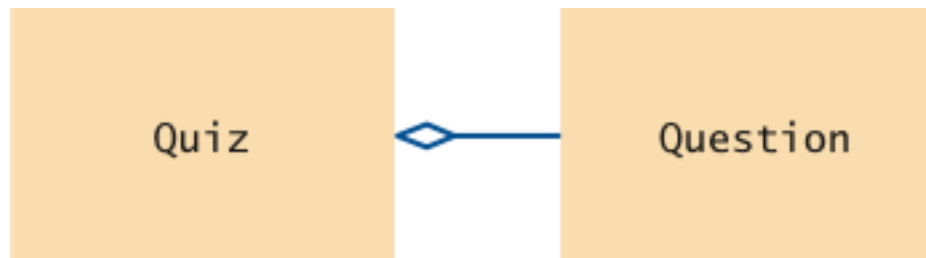
- ❑ If many classes depend on each other, the **coupling** between classes is high
- ❑ Good practice: minimize coupling between classes
 - Change in one class may require update of all coupled classes
 - Using a class in another program requires using all classes on which it depends

Coupling (2)



Aggregation (1)

- ❑ A class **aggregates** another of its objects contain objects of another class
 - “has-a” relationship
- ❑ Example: a quiz is made up of questions
 - Class `Quiz` aggregates class `Question`



Aggregation (2)

- ❑ Finding out about aggregation helps in implementing classes
- ❑ Example: since a quiz can have any number of questions, use an array or array list for collecting them

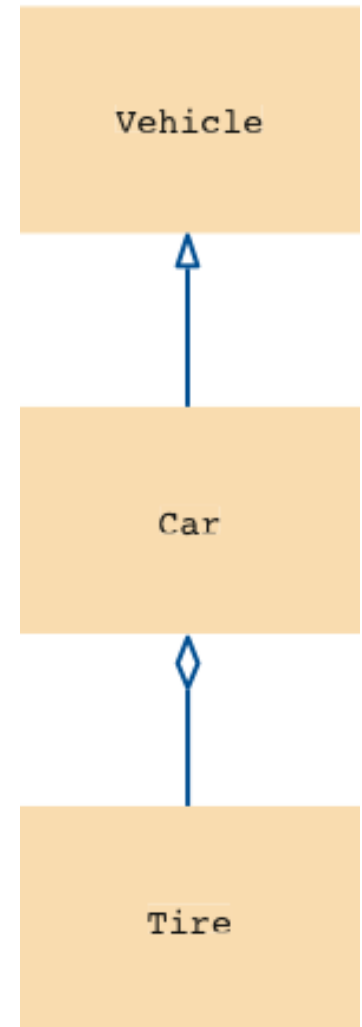
```
public class Quiz
{
    private ArrayList<Question> questions;
    . . .
}
```


Inheritance (1)

- ❑ **Inheritance** is the relationship between a more general class (**superclass**) and a more specialized class (**subclass**)
 - “is-a” relationship
- ❑ Example: every car *is a* vehicle; every car has tires
 - Class `Car` is a subclass of class `Vehicle`; class `car` aggregates class `Tire`





Inheritance (2)

```
public class Car extends Vehicle
{
    private Tire[] tires;
    . . .
}
```

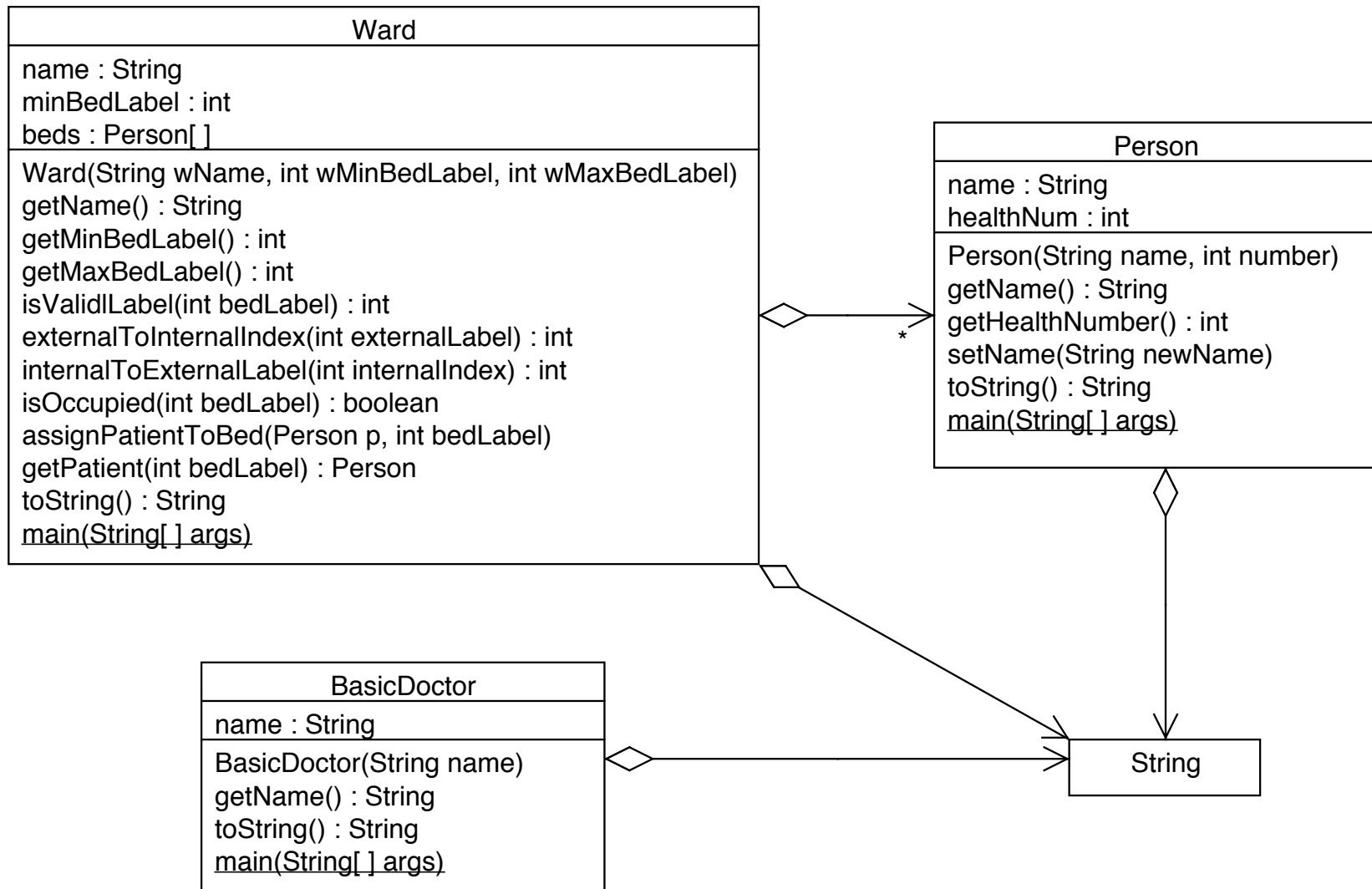


UML Relationship Symbols

Table 1 UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

UML Example: Assignment 2



UML Example: Assignment 2

Note..

The String class is a field or used by just about every class. As a result, it is often not shown in a class diagram. The students can omit it.

Note..

Obviously Ward uses the Person class, and they all use String. However, when a field or container aggregation relationship exists between two classes, it is assumed that there is also a uses relationship, so the uses relationship is not shown. Don't remove marks for including it.

Note..

A parameter for a constructor or method can appear as
type name (as in Java, and shown here)

or

name type (as is standard for UML).

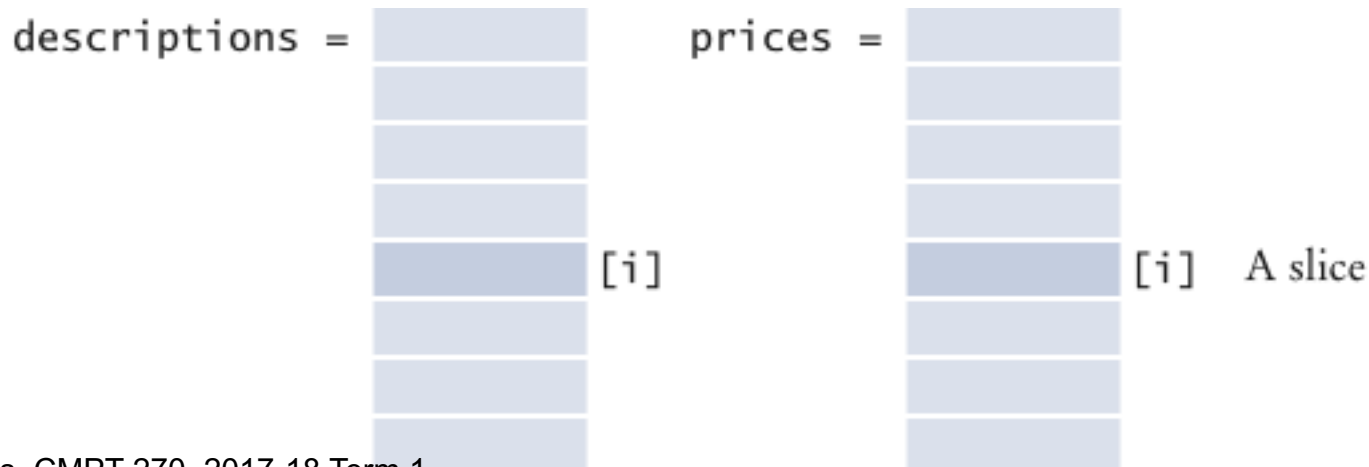
However, for fields, constants, constructors, methods, etc, the name should appear before other information as it makes it so much easier to find a name.

No marks off for the wrong order this time.

Parallel Arrays (1)

- ❑ **Parallel arrays** have the same length, each of which stores a part of what conceptually should be an object
- ❑ **Example:**

```
String[] descriptions;  
double[] prices;
```



Parallel Arrays (2)

- ❑ Programmer must ensure arrays always have the same length and that each slice is filled with values that belong together
- ❑ Any method that operates on a slice must get all values of the slice as parameters

Parallel Arrays (3)

- ❑ Avoid parallel arrays by changing them into an array of objects

- ❑ Example:

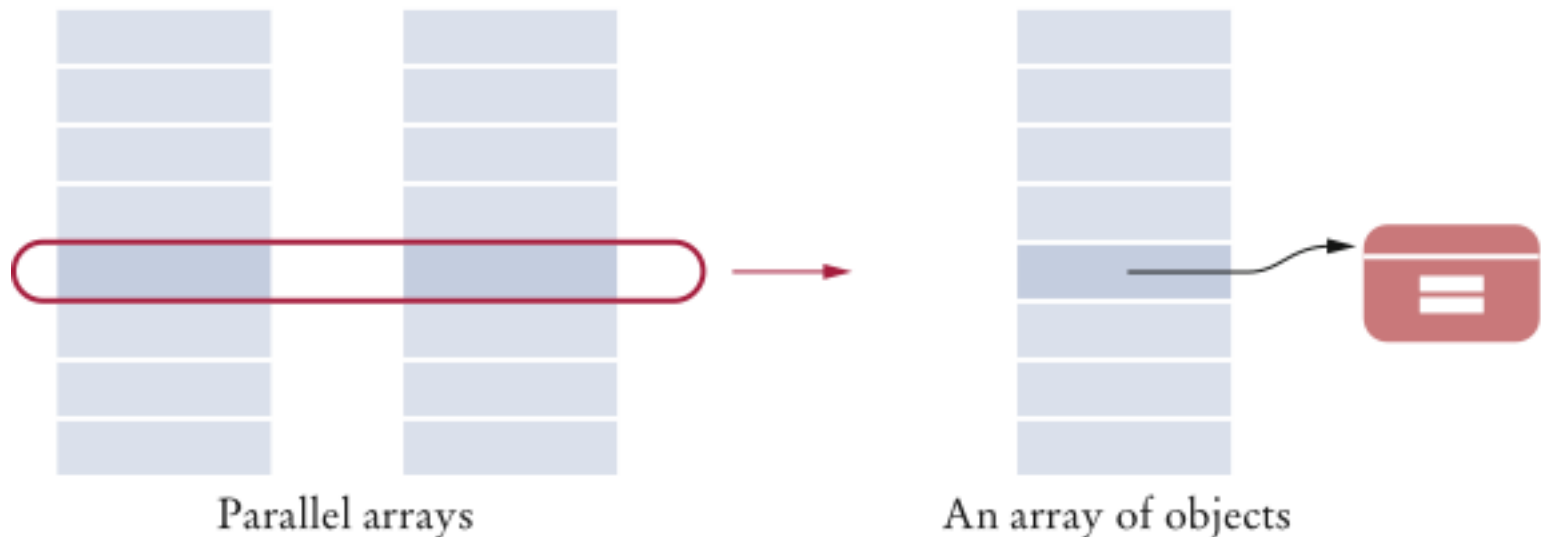
```
public class Item
{
    private String description;
    private double price;
}
```

- Replace parallel arrays with

```
Item[] items;
```


Parallel Arrays (4)

- Each slot in the resulting array corresponds to a slice in the set of parallel arrays



Summary

Discover Classes and their Responsibilities

- ❑ To discover classes, look for nouns in the problem description.
- ❑ Concepts from the problem domain are good candidates for classes.
- ❑ A CRC card describes a class, its responsibilities, and its collaborating classes
- ❑ The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

Summary

Class Relationships and UML Diagrams

- ❑ A class depends on another class if it uses objects of that class.
- ❑ It is a good practice to minimize the coupling (i.e., dependency) between classes.
- ❑ A class aggregates another if its objects contain objects of the other class.
- ❑ Inheritance (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate.
- ❑ Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.

Summary

- ❑ You need to be able to distinguish the UML notations for inheritance, interface implementation, aggregation, and dependency.
- ❑ Avoid parallel arrays by changing them into arrays of objects.

Summary

Object-Oriented Development Process

- ❑ Start the development process by gathering and documenting program requirements.
- ❑ Use CRC cards to find classes, responsibilities, and collaborators.
- ❑ Use UML diagrams to record class relationships.
- ❑ Use javadoc comments (with the method bodies left blank) to record the behavior of classes.
- ❑ After completing the design, implement your classes.