CHAPTER 12

OBJECT-ORIENTED DESIGN

Slides by Rick Giles

Chapter 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
- Packages

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**

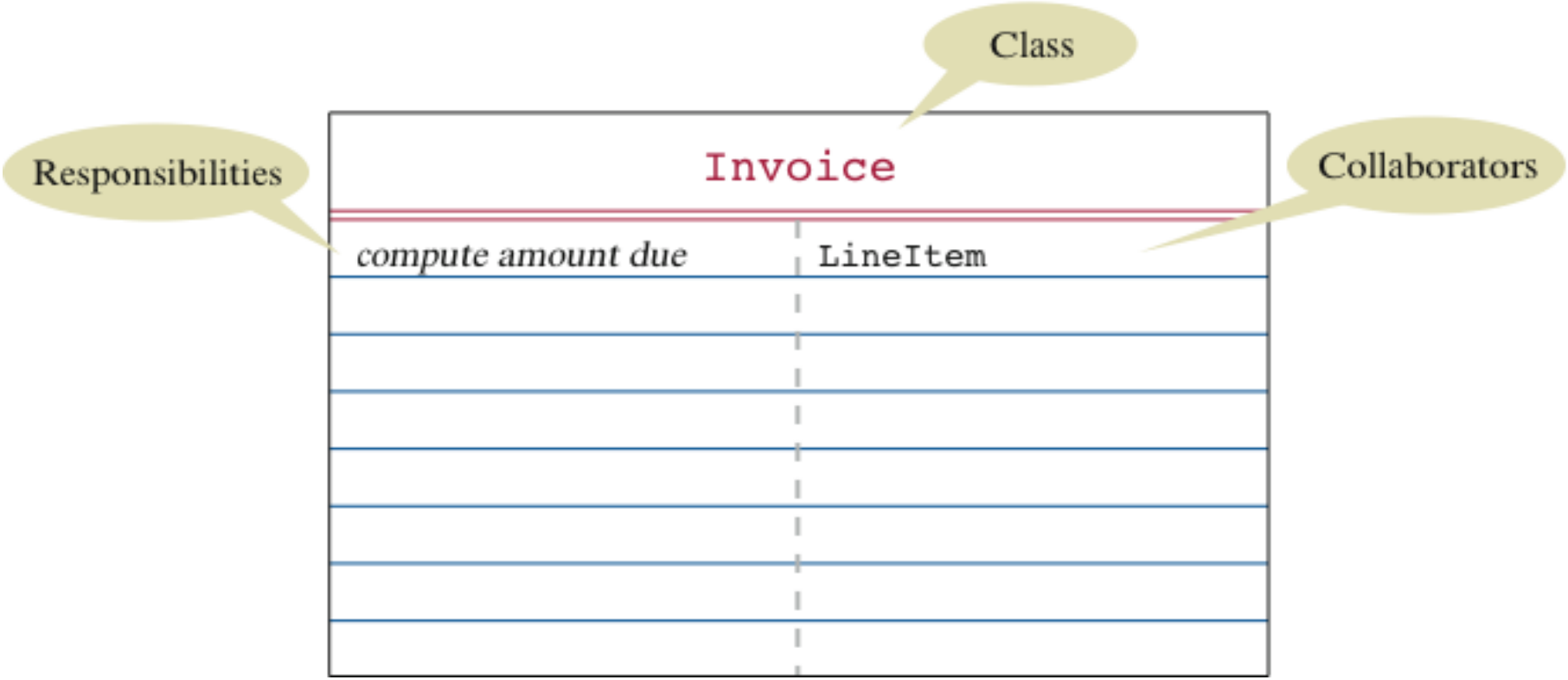
12.1 Classes and Their Responsibilities (2)

- 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:

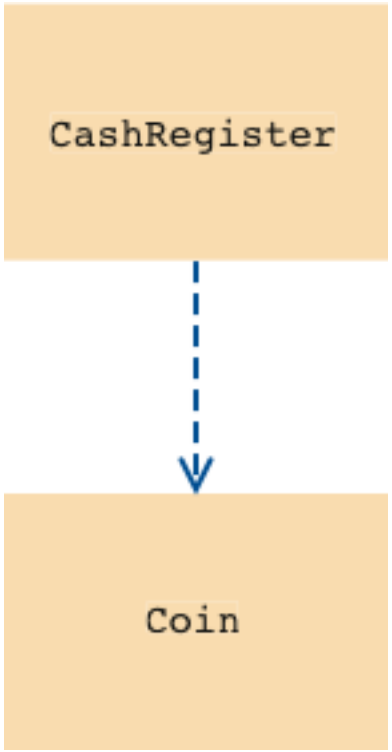
```
public class Coin
{
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}

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
- To visualize relationships, draw class diagrams
- **UML**: Unified Modeling Language
  - Notation for object-oriented analysis and design

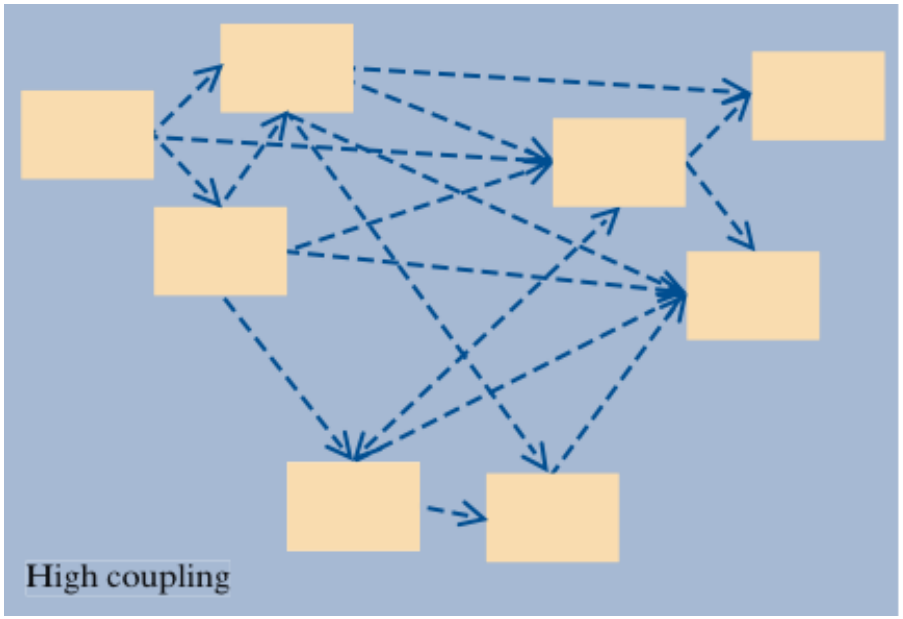
Dependency Relationship

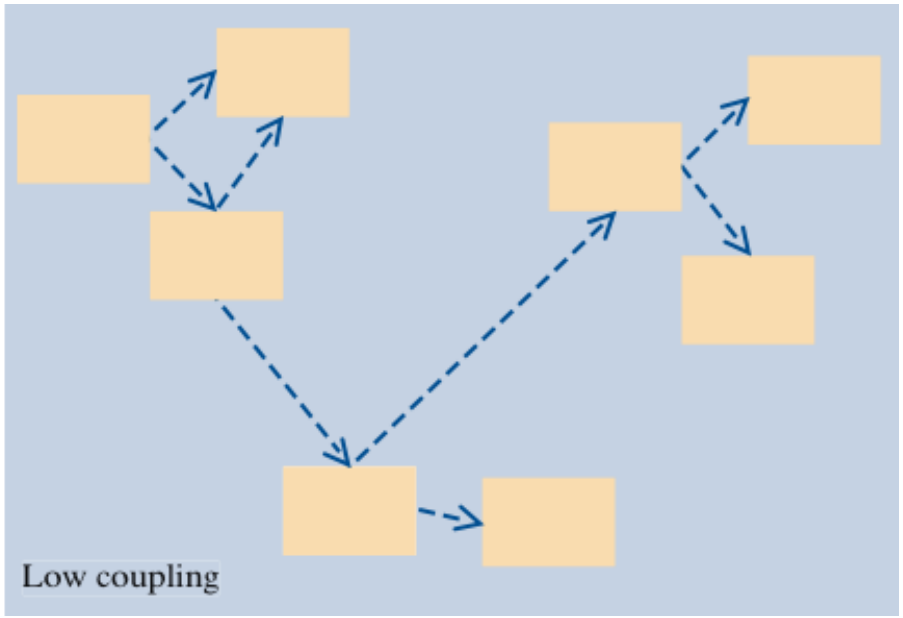


Coupling (1)

- 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

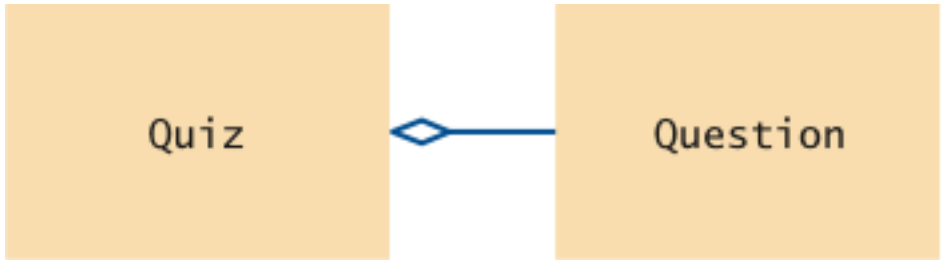
Coupling (2)





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

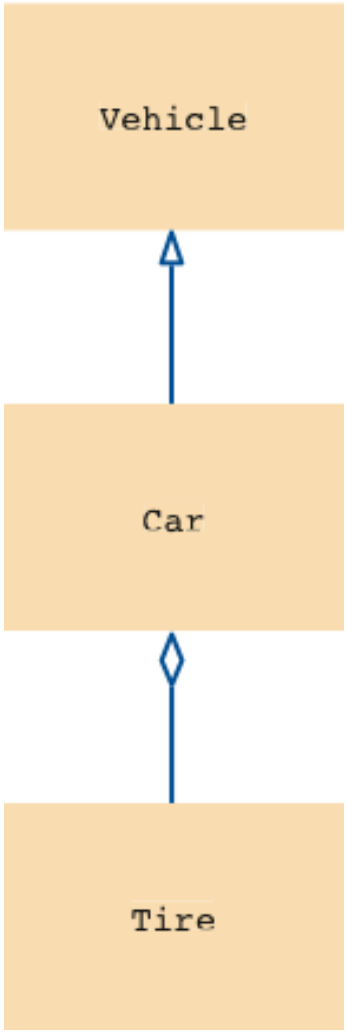
```
public class Quiz
{
   private ArrayList<Question> questions;
   . . .
}
```

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`

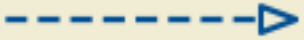
Inheritance (2)

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



UML Relationship Symbols

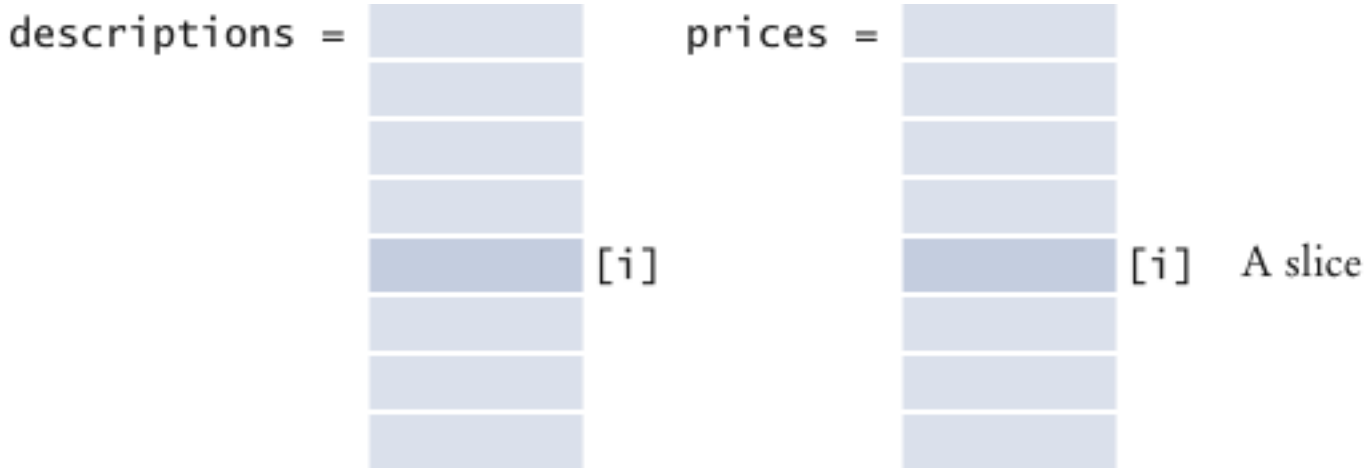
Table 1 UML Relationship Symbols

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

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:

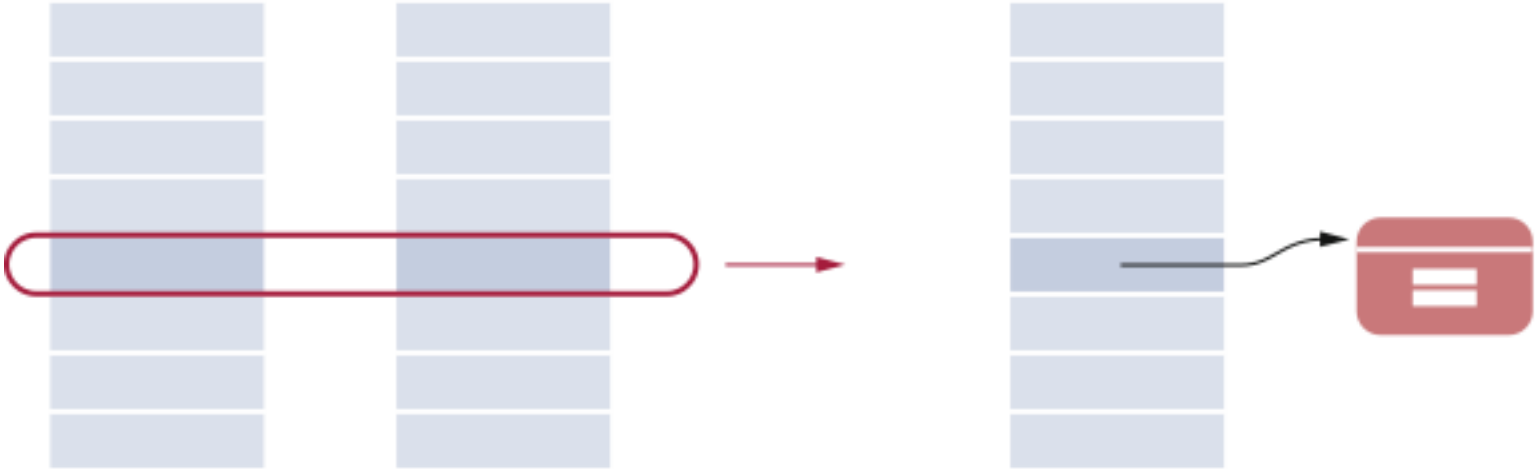
```
public class Item
{
   private String description;
   private double price;
}
```

  - 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



12.3 Application: Printing an Invoice

- Five-part development process:
  1. Gather requirements.
  2. Use CRC cards to find classes, responsibilities, collaborators.
  3. Use UML diagrams to record relationships.
  4. Use `javadoc` to document method behavior.
  5. Implement your program.

Requirements

- Program prints the billing address, all line items, and the amount due

```
                    I N V O I C E

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

Description                    Price  Qty  Total
Toaster                        29.95   3   89.85
Hair dryer                     24.95   1   24.95
Car vacuum                     19.99   2   39.98

AMOUNT DUE: $154.78
```

CRC Cards (1)

- Nouns from requirements:

```
Invoice                 Product
LineItem                Price
Description             Total
Quantity
Amount due
Address
```

- `Description` and `Price` are attributes of the `Product` class
- `Quantity` is an attribute of the `LineItem` class
- `Total` and `Amount due` are computed

CRC Cards (2)

- Left with four candidate classes:

```
Invoice
Address
LineItem
Product
```

CRC Cards (3)

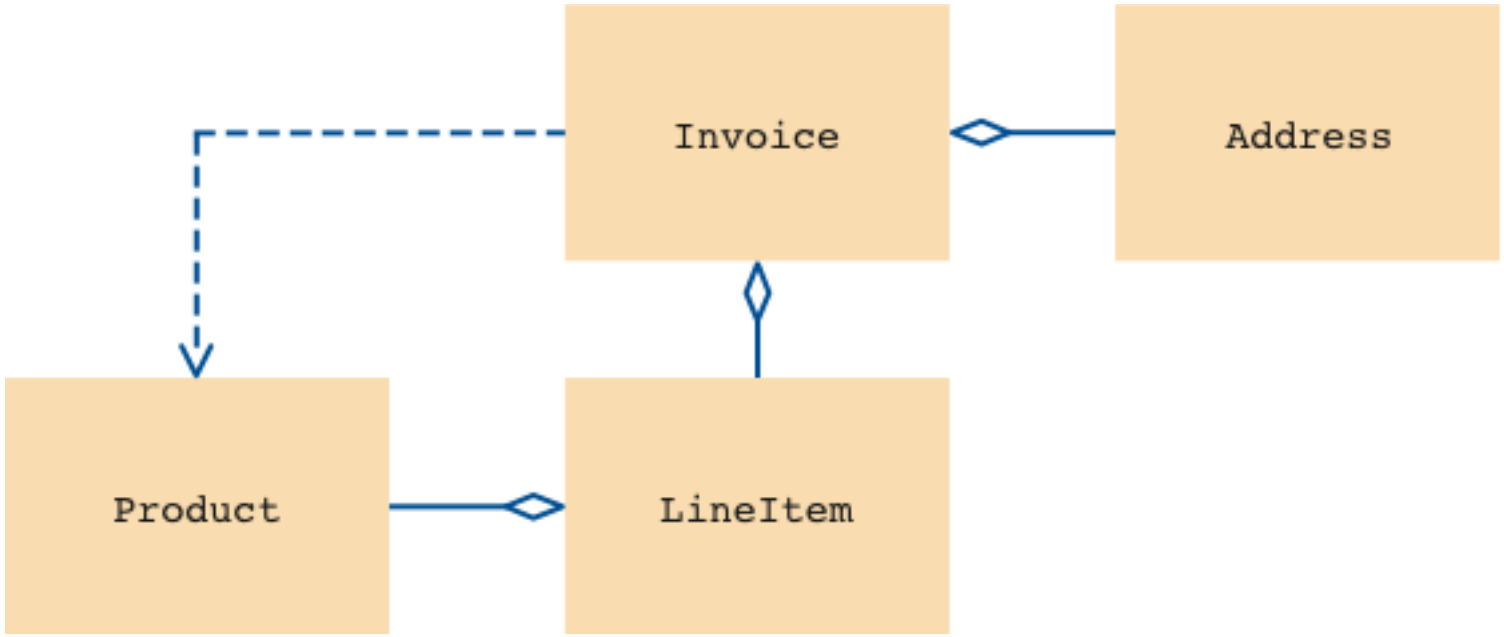
Address
format the address

Product
get description
get unit price

LineItem	
format the item	`Product`
get total price	

Invoice	
format the invoice	`Address`
add a product and quantity	`LineItem`
	`Product`

UML Class Diagram



Method Documentation (1)

```
/**
   Describes an invoice for a set of purchased products.
*/
public class Invoice
{
   /**
      Adds a charge for a product to this invoice.
      @param aProduct the product that the customer ordered
      @param quantity the quantity of the product
   */
   public void add(Product aProduct, int quantity)
   {
   }

   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
   }
}
```

Method Documentation (2)

```
/**
   Describes a quantity of an article to purchase.
*/
public class LineItem
{
   /**
      Computes the total cost of this line item.
      @return the total price
   */
   public double getTotalPrice()
   {
   }

   /**
      Formats this item.
      @return a formatted string of this item
   */
   public String format()
   {
   }
}
```

Method Documentation (3)

```
/**
   Describes a product with a description and a price.
*/
public class Product
{
   /**
      Gets the product description.
      @return the description
   */
   public String getDescription()
   {
   }

   /**
      Gets the product price.
      @return the unit price
   */
   public double getPrice()
   {
   }
}
```

Method Documentation (4)

```
/**
   Describes a mailing address.
*/
public class Address
{
   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
   }
}
```

Class Documentation in HTML Format

Invoice - Firefox

File Edit View History Bookmarks Tools Help

file:///home/cay/BigJava/ch12/invoice/index.html

All Classes
Address
Invoice
InvoicePrinter
LineItem
Product

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Invoice

```
java.lang.Object
  └ Invoice
```

```
public class Invoice
extends java.lang.Object
```

Describes an invoice for a set of purchased products.

Constructor Summary

Invoice(Address anAddress)	Constructs an invoice.

Method Summary

void	add(Product aProduct, int quantity) Adds a charge for a product to this invoice.
java.lang.String	format() Formats the invoice.
double	getAmountDue() Computes the total amount due.

InvoicePrinter.java

```
1   /**
2      This program demonstrates the invoice classes by printing
3      a sample invoice.
4   */
5   public class InvoicePrinter
6   {
7      public static void main(String[] args)
8      {
9         Address samsAddress
10              = new Address("Sam&apos;s Small Appliances",
11                 "100 Main Street", "Anytown", "CA", "98765");
12
13        Invoice samsInvoice = new Invoice(samsAddress);
14        samsInvoice.add(new Product("Toaster", 29.95), 3);
15        samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16        samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17
18        System.out.println(samsInvoice.format());
19     }
20  }
```

Invoice.java

```
1  import java.util.ArrayList;
2
3  /**
4     Describes an invoice for a set of purchased products.
5  */
6  public class Invoice
7  {
8     private Address billingAddress;
9     private ArrayList<LineItem> items;
10
11    /**
12       Constructs an invoice.
13       @param anAddress the billing address
14    */
15    public Invoice(Address anAddress)
16    {
17       items = new ArrayList<LineItem>();
18       billingAddress = anAddress;
19    }
20
```

Continued

Invoice.java (cont.)

```
21      /**
22         Adds a charge for a product to this invoice.
23         @param aProduct the product that the customer ordered
24         @param quantity the quantity of the product
25      */
26      public void add(Product aProduct, int quantity)
27      {
28         LineItem anItem = new LineItem(aProduct, quantity);
29         items.add(anItem);
30      }
31
```

Continued

Invoice.java (cont.)

```
32      /**
33         Formats the invoice.
34         @return the formatted invoice
35      */
36      public String format()
37      {
38         String r =  "                          I N V O I C E\n\n"
39               + billingAddress.format()
40               + String.format("\n\n%-30s%8s%5s%8s\n",
41                  "Description", "Price", "Qty", "Total");
42
43         for (LineItem item : items)
44         {
45            r = r + item.format() + "\n";
46         }
47
48         r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());
49
50         return r;
51      }
52
```

Continued

Invoice.java (cont.)

```
53      /**
54         Computes the total amount due.
55         @return the amount due
56      */
57      public double getAmountDue()
58      {
59         double amountDue = 0;
60         for (LineItem item : items)
61         {
62            amountDue = amountDue + item.getTotalPrice();
63         }
64         return amountDue;
65      }
66   }
```

LineItem.java

```
1  /**
2     Describes a quantity of an article to purchase.
3  */
4  public class LineItem
5  {
6     private int quantity;
7     private Product theProduct;
8
9     /**
10       Constructs an item from the product and quantity.
11       @param aProduct the product
12       @param aQuantity the item quantity
13    */
14    public LineItem(Product aProduct, int aQuantity)
15    {
16       theProduct = aProduct;
17       quantity = aQuantity;
18    }
19
```

Continued

LineItem.java (cont.)

```
20      /**
21         Computes the total cost of this line item.
22         @return the total price
23      */
24      public double getTotalPrice()
25      {
26         return theProduct.getPrice() * quantity;
27      }
28
29      /**
30         Formats this item.
31         @return a formatted string of this item
32      */
33      public String format()
34      {
35         return String.format("%-30s%8.2f%5d%8.2f",
36            theProduct.getDescription(), theProduct.getPrice(),
37            quantity, getTotalPrice());
38      }
39   }
```

Product.java

```
1  /**
2     Describes a product with a description and a price.
3  */
4  public class Product
5  {
6     private String description;
7     private double price;
8
9     /**
10       Constructs a product from a description and a price.
11       @param aDescription the product description
12       @param aPrice the product price
13    */
14    public Product(String aDescription, double aPrice)
15    {
16       description = aDescription;
17       price = aPrice;
18    }
19
```

Continued

Product.java (cont.)

```
20      /**
21         Gets the product description.
22         @return the description
23      */
24      public String getDescription()
25      {
26         return description;
27      }
28
29      /**
30         Gets the product price.
31         @return the unit price
32      */
33      public double getPrice()
34      {
35         return price;
36      }
37   }
```

Address.java

```
1   /**
2      Describes a mailing address.
3   */
4   public class Address
5   {
6      private String name;
7      private String street;
8      private String city;
9      private String state;
10     private String zip;
11
12     /**
13        Constructs a mailing address.
14        @param aName the recipient name
15        @param aStreet the street
16        @param aCity the city
17        @param aState the two-letter state code
18        @param aZip the ZIP postal code
19     */
```

Continued

Address.java (cont.)

```
20      public Address(String aName, String aStreet,
21            String aCity, String aState, String aZip)
22      {
23         name = aName;
24         street = aStreet;
25         city = aCity;
26         state = aState;
27         zip = aZip;
28      }
29
30      /**
31         Formats the address.
32         @return the address as a string with three lines
33      */
34      public String format()
35      {
36         return name + "\n" + street + "\n"
37               + city + ", " + state + " " + zip;
38      }
39   }
```

12.4 Packages

- **Package**: a set of related classes
- Important packages in the Java library:

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database Access	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`omg.w3c.dom`	Document Object Model for XML documents	`Document`

Organizing Related Classes into Packages (1)

- To put a class in a package, you must place

```
package packageName;
```

  as the first statement in its source

- Package name consists of one or more identifiers separated by periods

Organizing Related Classes into Packages (2)

- For example, to put the `BankAccount` class into a package named `com.horstmann`, the `BankAccount.java` file must start as follows:

```
package com.horstmann;

public class BankAccount
{
   . . .
}
```

- **Default package** has no name, no `package` statement

Importing Packages

- Can always use class without importing:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Tedious to use fully qualified name
- Import lets you use shorter class name:

```
import java.util.Scanner;
...
Scanner in = new Scanner(System.in);
```

- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import classes in package `java.lang`
- Don’t need to import other classes in the same package

Package Names

- Use packages to avoid name clashes

  `java.util.Timer`

  vs.

  `javax.swing.Timer`

- Package names should be unambiguous
- Recommendation: start with reversed domain name:

  `com.horstmann`

- `edu.sjsu.cs.walters`: for Britney Walters' classes (`walters@cs.sjsu.edu`)

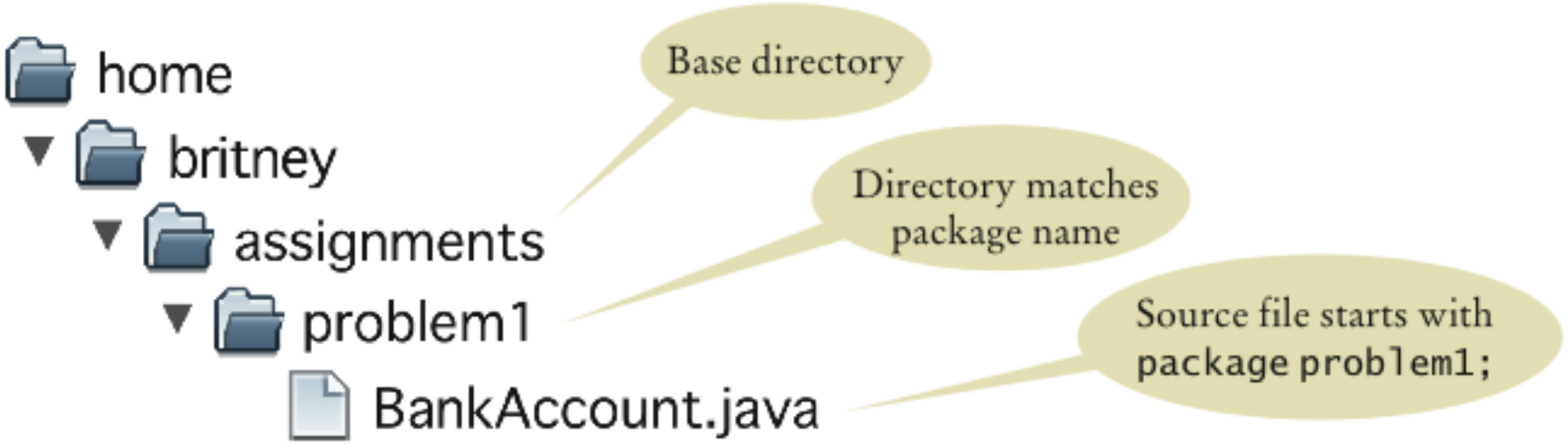
How Classes Are Located

- **Base directory:** holds your program's source files
- Path of a class source file, relative to base directory, must match its package name
- Example: if base directory is

  `/home/britney/assignments`

  place source files for classes in package `problem1` in directory

  `/homehome/britney/assignments/problem1`



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.

Summary

Packages

- A package is a set of related classes.
- Use packages to structure the classes in your program.
- The `import` directive lets you refer to a class from a package by its class name, without the package prefix.