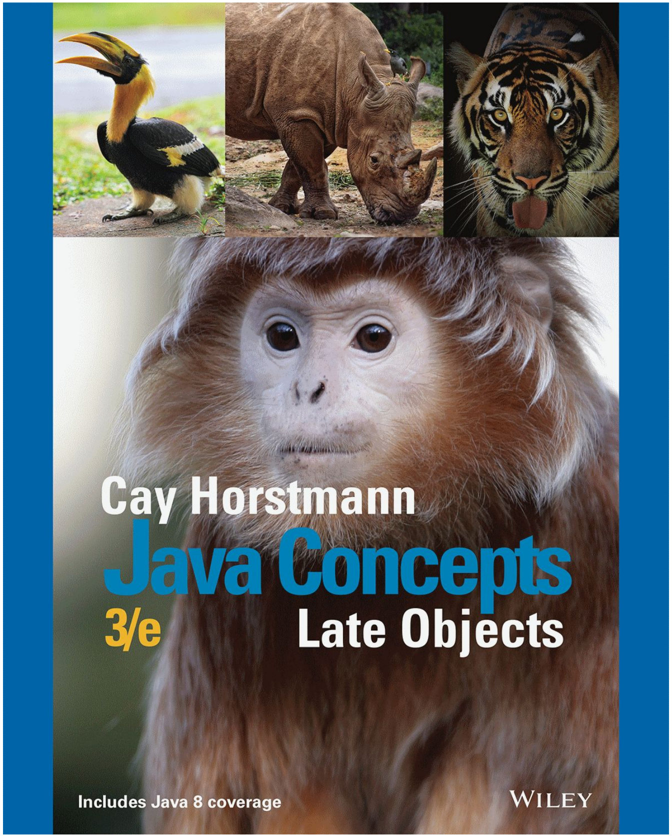
Chapter 12 - Object-Oriented Design



Chapter Goals

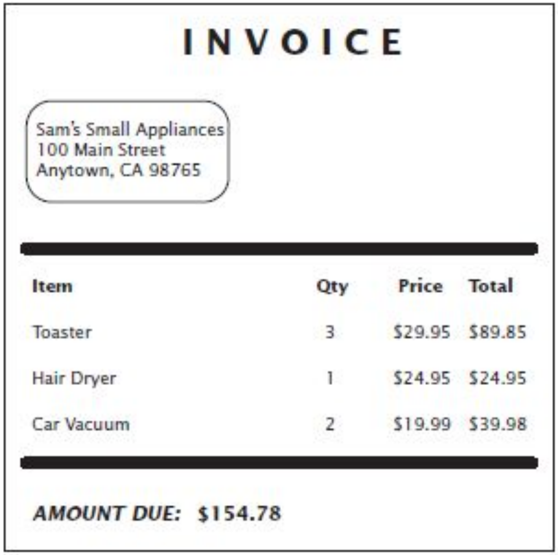
© Petrea Alexandra/iStockphoto.

- To learn how to discover new classes and methods To use CRC cards for class discovery
- 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

Discovering Classes

- When designing a program, you work from a requirements specification
  - The designer’s task is to discover structures that make it possible to implement the requirements
- To discover classes, look for nouns in the problem description.
- Find methods by looking for verbs in the task description.

Example: Invoice



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**

Figure 1 An Invoice

Example: Invoice

- Classes that come to mind:
  - `Invoice`
  - `LineItem`
  - `Customer`
- Good idea to keep a list of candidate classes.
- Brainstorm: put all ideas for classes onto the
- list. Cross not useful ones later.
- Concepts from the problem domain are good candidates for classes.
- Not all classes can be discovered from the program requirements:
  - Most programs need tactical classes

The CRC Card Method

© Oleg Prikhodko/iStockphoto.

In a class scheduling system, potential classes from the problem domain include Class, LectureHall, Instructor, and Student.

The CRC Card Method

- After you have a set of classes
  - Define the behavior (methods) of each class
- Look for verbs in the task description
  - Match the verbs to the appropriate objects
- The invoice program needs to compute the amount due
  - Which class is responsible for this method?
    - Invoice class

The CRC Card Method

- To find the class responsibilities, use the CRC card method.
- A CRC card describes a class, its responsibilities, and its collaborating classes.
  - CRC - stands for “classes”, “responsibilities”, “collaborators”
- Use an index card for each class.
- Pick the class that should be responsible for each method (verb).
- Write the responsibility onto the class card.
- Indicate what other classes are needed to fulfill responsibility (collaborators).

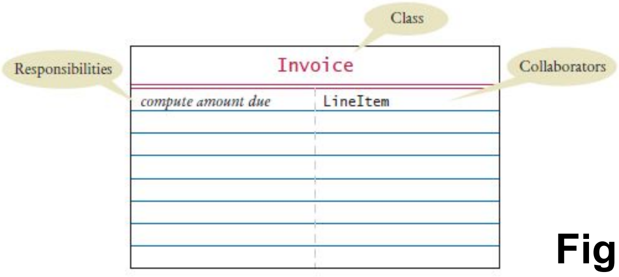


Figure 2 A CRC Card

Self Check 12.1

What is the rule of thumb for finding classes?

Answer: Look for nouns in the problem description.

Self Check 12.2

Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?

Answer: Yes (`ChessBoard`) and no (`MovePiece`).

Self Check 12.3

Suppose the invoice is to be saved to a file. Name a likely collaborator.

Answer: `PrintStream`

Self Check 12.4

Looking at the invoice in Figure 1, what is a likely responsibility of the `Customer` class?

Answer: To produce the shipping address of the customer.

Self Check 12.5

What do you do if a CRC card has ten responsibilities?

Answer: Reword the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.

Relationships Between Classes

The most common types of relationships:

- Dependency
- Aggregation
- Inheritance

Dependency

- A class depends on another class if it uses objects of that class.
  - *The "knows about" relationship.*
- Example: `CashRegister` depends on `Coin`

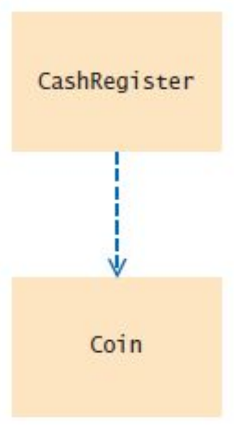


Figure 3 Dependency Relationship Between the `CashRegiste` and `Coin` Classes

Dependency

- It is a good practice to minimize the coupling (i.e., dependency) between classes.

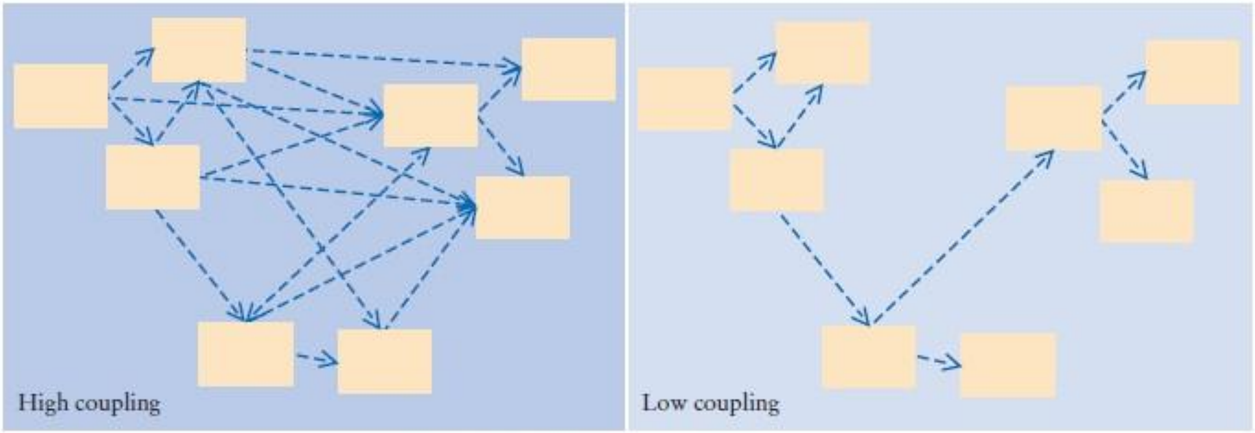


Figure 4 High and Low Coupling Between Classes

- When a class changes, coupled classes may also need updating.

Aggregation

- A class aggregates another if its objects contain objects of the other class.
    - *Has-a* relationship
- Example: a `Quiz` class aggregates a `Question`
- class. The UML for aggregation:

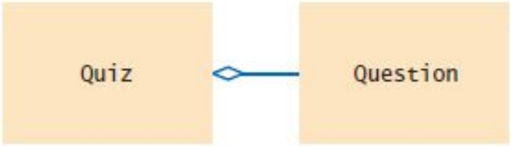


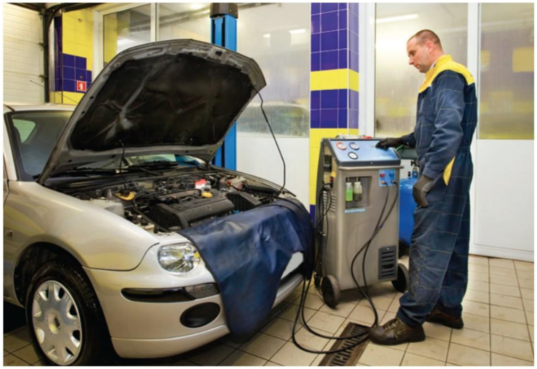
Figure 5
Class Diagram
Showing Aggregation

- Aggregation is a stronger form of dependency.
- Use aggregation to remember another object between method calls.
- Use an instance variable

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

- A class may use the `Scanner` class without ever declaring an instance variable of class `Scanner`.
    - This is dependency NOT aggregation

Aggregation



© bojan fatur/iStockphoto.

A car has a motor and tires. In object-oriented design, this “has-a” relationship is called aggregation.

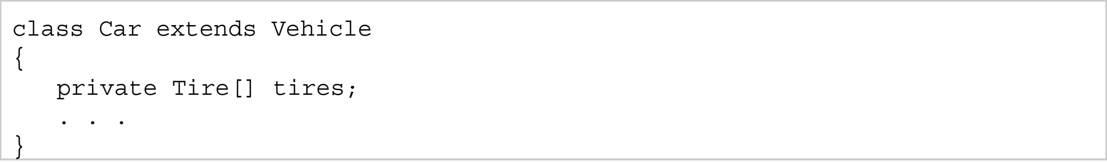
Inheritance

- Inheritance is a relationship between a more general class (the superclass) and a more specialized class (the subclass).
  - The “is-a” relationship.
  - Example: Every truck is a vehicle.
- Inheritance is sometimes inappropriately used when the has-a relationship would be more appropriate.
  - Should the class `Tire` be a subclass of a class `Circle`? No
    - A tire has a circle as its boundary
    - Use aggregation

```
public class Tire
{
   private String rating;
   private Circle boundary;
   . . .
}
```

Inheritance

- Every car is a vehicle. (Inheritance)
- Every car has a tire (or four). (Aggregation)



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

- Aggregation denotes that objects of one class contain references to objects of another class.

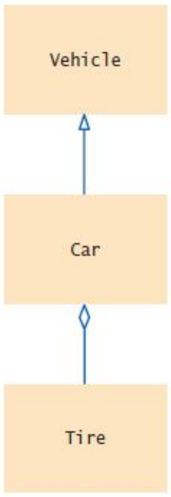
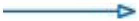
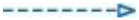
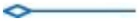
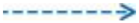


Figure 6 UML Notation for Inheritance and Aggregation

UML Relationship Symbols

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

Self Check 12.6

Consider the `CashRegisterTester` class of Section 8.2. On which classes does it depend?

> **Answer:** The `CashRegisterTester` class depends on the `CashRegister`, `Coin`, and `System` classes.

Self Check 12.7

Consider the `Question` and `ChoiceQuestion` objects of Chapter 9. How are they related?

> **Answer:** The `ChoiceQuestion` class inherits from the `Question` class.

Self Check 12.8

Consider the `Quiz` class described in Section 12.2.2. Suppose a `quiz` contains a mixture of `Question` and `ChoiceQuestion` objects. Which classes does the `Quiz` class depend on?

Answer: The `Quiz` class depends on the `Question` class but probably not `ChoiceQuestion`, if we assume that the methods of the `Quiz` class manipulate generic `Question` objects, as they did in Chapter 9.

Self Check 12.9

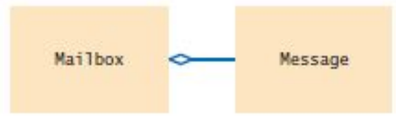
Why should coupling be minimized between classes?

Answer: If a class doesn’t depend on another, it is not affected by interface changes in the other class.

Self Check 12.10

In an e-mail system, messages are stored in a mailbox. Draw a UML diagram that shows the appropriate aggregation relationship.

Answer:



Self Check 12.11

You are implementing a system to manage a library, keeping track of which books are checked out by whom. Should the `Book` class aggregate `Patron` or the other way around?

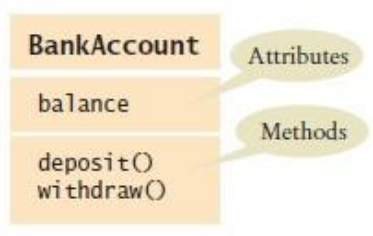
Answer: Typically, a library system wants to track which books a patron has checked out, so it makes more sense to have `Patron` aggregate `Book`. However, there is not always one true answer in design. If you feel strongly that it is important to identify the patron who checked out a particular book (perhaps to notify the patron to return it because it was requested by someone else), then you can argue that the aggregation should go the other way around.

Self Check 12.12

In a library management system, what would be the relationship between classes `Patron` and `Author`?

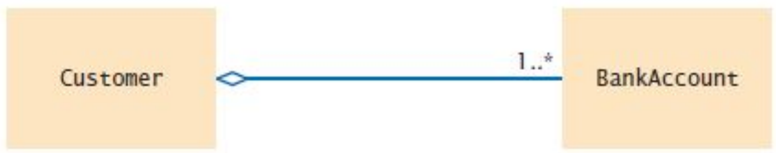
Answer: There would be no relationship.

Attributes and Methods in UML Diagrams



Multiplicities

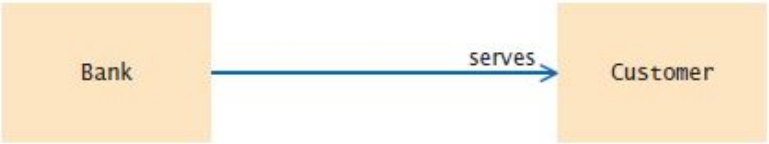
- any number (zero or more): *
- one or more: 1..*
- zero or one: 0..1
- exactly one: 1



- *An Aggregation Relationship with Multiplicities*

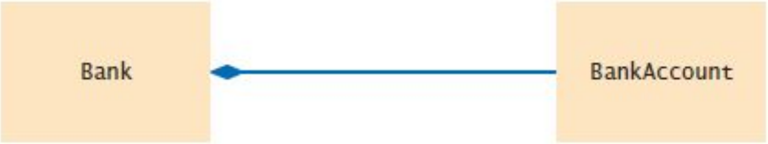
Aggregation and Association, and Composition

- Association: More general relationship between classes.
- Use early in the design phase.
- A class is associated with another if you can navigate from objects of one class to objects of the other.
- Given a `Bank` object, you can navigate to `Customer` objects.



An Association Relationship

- Composition: one of the classes can not exist without the otl



A Composition Relationship

Application: Printing an Invoice

Five-part program development process

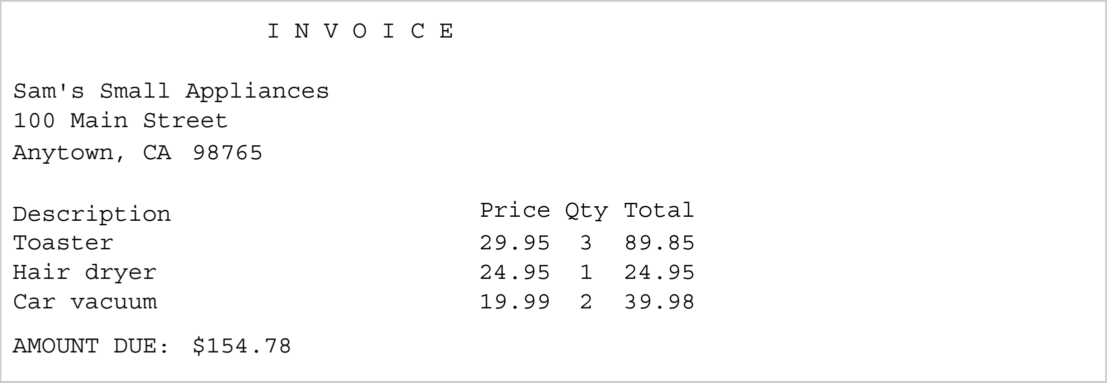
1. Gather requirements
2. Use CRC cards to find classes, responsibilities, and collaborators
3. Use UML diagrams to record class relationships
4. Use `javadoc` to document method behavior
5. Implement your program

Application: Printing an Invoice — Requirements

- Start the development process by gathering and documenting program requirements.
- Task: Print out an invoice
- Invoice: Describes the charges for a set of products in certain quantities.
- Omit complexities
  - Dates, taxes, and invoice and customer numbers
- Print invoice
  - Billing address, all line items, amount due
- Line item
  - Description, unit price, quantity ordered, total price
- For simplicity, do not provide a user interface.
- Test program: Adds line items to the invoice and then prints it.

Application: Printing an Invoice

- Sample Invoice



```
                  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
```

- An invoice lists the charges for each item and the amou

© Scott Cramer/iStockphoto.

Application: Printing an Invoice — CRC Cards

- Use CRC cards to find classes, responsibilities, and collaborators.
- Discover classes
- Nouns are possible classes:

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

Application: Printing an Invoice — CRC Cards

- Analyze classes:

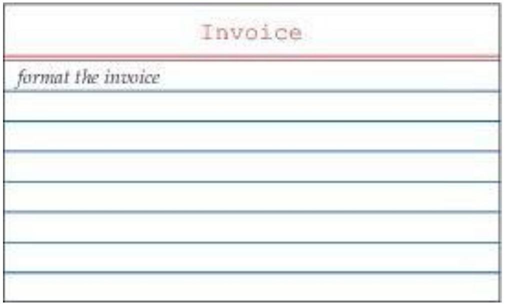
```
Invoice
Address
LineItem    // Records the product and the quantity
Product
Description // Field of the Product class
Price // Field of the Product class  Quantity
// Not an attribute of a Product  Total
// Computed — not stored anywhere  Amount Due
// Computed — not stored anywhere
```

- Classes after a process of elimination:

```
Invoice
Address
LineItem
Product
```

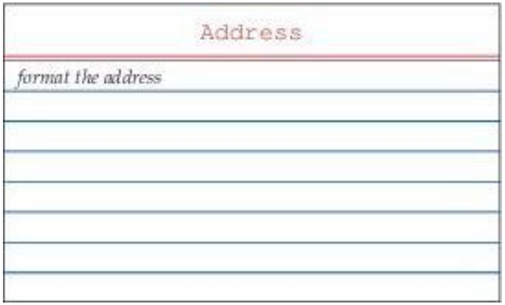
CRC Cards for Printing Invoice

`Invoice` and `Address` must be able to format themselves:



Invoice

format the invoice

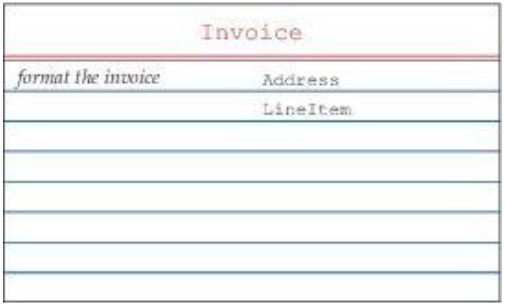


Address

format the address

CRC Cards for Printing Invoice

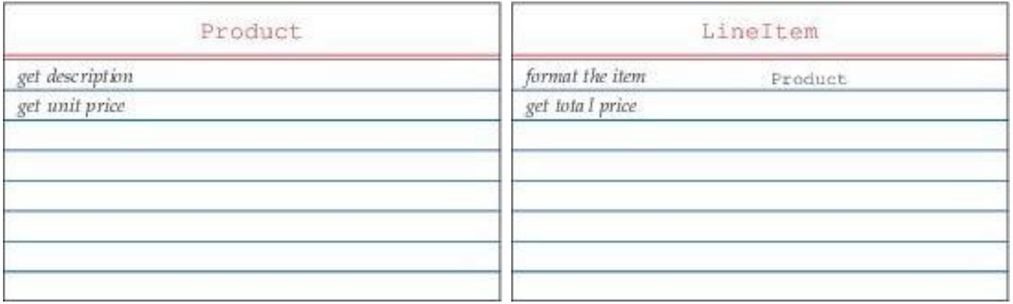
Add collaborators to `Invoice` card:



Invoice	
format the invoice	Address
	LineItem

CRC Cards for Printing Invoice

Product and LineItem CRC cards:

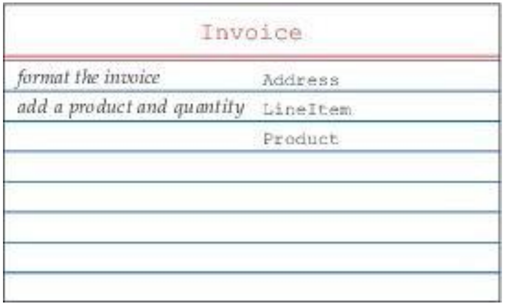


Product	
get description	
get unit price	

LineItem	
format the item	Product
get tota l price	

CRC Cards for Printing Invoice

`Invoice` must be populated with products and quantities:



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

Application: Printing an Invoice — UML Diagrams

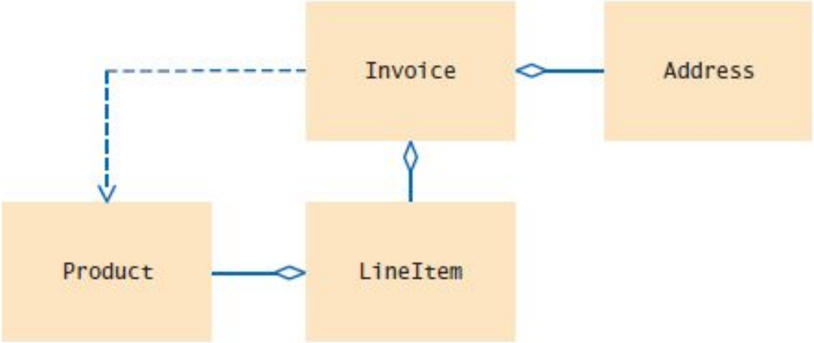


Figure 7 The Relationships Between the Invoice Classes

Printing an Invoice — Method Documentation

- Use `javadoc` comments (with the method bodies left blank) to record the behavior of the classes.
- Write a Java source file for each class:

    Write the method comments for those methods that you have discovered,

    Leave the body of the methods blank

- Run `javadoc` to obtain formatted version of documentation in HTML format.
- Advantages:

    Share HTML documentation with other team members

    Format is immediately useful: Java source files

    Supply the comments of the key methods

Method Documentation — Invoice Class

```
/**
   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 — LineItem Class

```
/**
   Describes a quantity of an article to purchase and its price.
*/
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 line item
   */
   public String format()
   {
   }
}
```

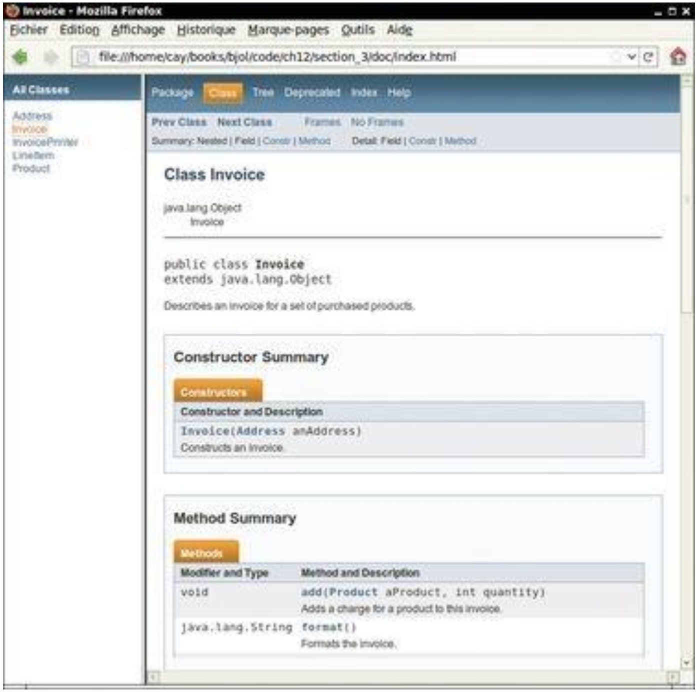
Method Documentation — Product Class

```
/**
   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 — Address Class

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

The Class Documentation in the HTML Format



Invoice - Mozilla Firefox
Fichier Edition Affichage Historique Marque-pages Outils Aide
file:///home/cay/books/bjol/code/ch12/section_3/doc/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

Constructors

Constructor and Description
`Invoice(Address anAddress)` Constructs an Invoice.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`add(Product aProduct, int quantity)` Adds a charge for a product to this invoice.
`java.lang.String`	`format()` Formats the invoice.

Figure 8 Class Documentation in HTML Format

Printing an Invoice — Implementation

- After completing the design, implement your
- classes. The UML diagram will give instance variables:

    Look for aggregated classes

    They yield instance variables

Implementation

- Invoice aggregates Address and LineItem. Every invoice has one billing address.
- An invoice can have many line items:

```
public class Invoice
{
   . . .
   private Address billingAddress;
   private ArrayList<LineItem> items;
}
```

Implementation

A line item needs to store a `Product` object and quantity:

```
public class LineItem
{
   . . .
   private int quantity;
   private Product theProduct;
}
```

Implementation

- The methods themselves are now very
- easy. Example:

getTotalPrice of LineItem gets the unit price of the product and multiplies it with the quantity

```
/**
   Computes the total cost of this line item.
   @return the total price
*/
public double getTotalPrice()
{
   return theProduct.getPrice() * quantity;
}
```

- Also supply constructors

section_3/InvoicePrinter.java

```java
/**
   This program demonstrates the invoice classes by printing
   a sample invoice.
*/
public class InvoicePrinter
{
public static void main(String[] args)
{
Address samsAddress
```

section_3/Invoice.java

```java
import java.util.ArrayList;

/**
Describes an invoice for a set of purchased products.
*/
public class Invoice
{
private Address billingAddress;
private ArrayList<LineItem> items;
```

section_3/LineItem.java

```
 /**
 Describes a quantity of an article to purchase.
 */
 public class LineItem
 {
private int quantity;
private Product theProduct;
/*
*
```

section_3/Product.java

```
 /**
 Describes a product with a description and a price.
 */
 public class Product
 {
private String description;
private double price;
/*
*
```

section_3/Address.java

```java
/**
Describes a mailing address.
*/
public class Address
{
    private String name;
    private String street;
    private String city;
    private String state;
```

Self Check 12.13

Which class is responsible for computing the amount due? What are its collaborators for this task?

> **Answer:** The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.

Self Check 12.14

Why do the format methods return `String` objects instead of directly printing to `System.out`?

Answer: This design decision reduces coupling. It enables us to reuse the classes when we want to show the invoice in a dialog box or on a web page.