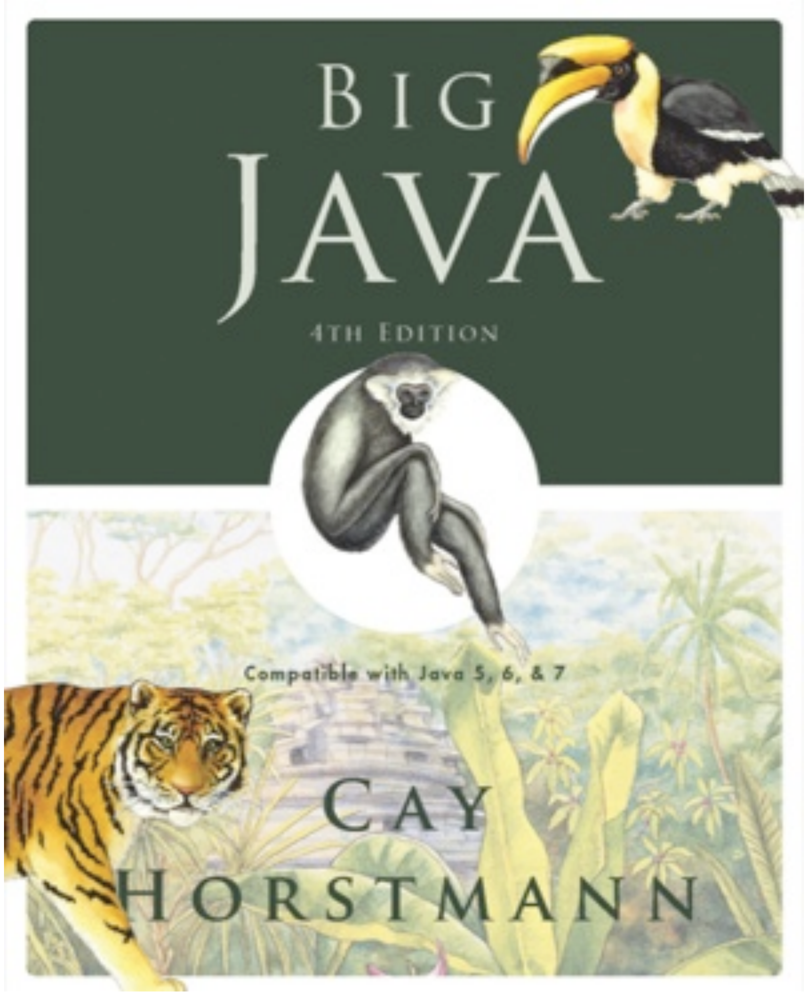


Chapter 12 – Object-Oriented Design

Big Java by Cay Horstmann

Chapter Goals

- To learn about the software life cycle
- To learn how to discover new classes and methods
- To understand the use of CRC cards for class discovery
- To be able to identify inheritance, aggregation, and dependency relationships between classes
- To master the use of UML class diagrams to describe class relationships
- To learn how to use object-oriented design to build complex programs

Big Java by Cay Horstmann

The Software Life Cycle

- Encompasses all activities from initial analysis until obsolescence
- Formal process for software development
  - *Describes phases of the development process*
  - *Gives guidelines for how to carry out the phases*
- Development process
  - *Analysis*
  - *Design*
  - *Implementation*
  - *Testing*
  - *Deployment*

Analysis

- Decide what the project is supposed to do
- Do not think about how the program will accomplish tasks
- Output: Requirements document
    - *Describes what program will do once completed*
    - *User manual: Tells how user will operate program*
    - *Performance criteria*

Design

- Plan how to implement the system
- Discover structures that underlie problem to be solved
- Decide what classes and methods you need
- Output:
  - *Description of classes and methods*
  - *Diagrams showing the relationships among the classes*

Implementation

- Write and compile the code
- Code implements classes and methods discovered in the design phase
- Program Run: Completed program

Testing

- Run tests to verify the program works correctly
- Program Run: A report of the tests and their results

Deployment

- Users install program
- Users use program for its intended purpose

The Waterfall Model

- Sequential process of analysis, design, implementation, testing, and deployment
- When rigidly applied, waterfall model did not work

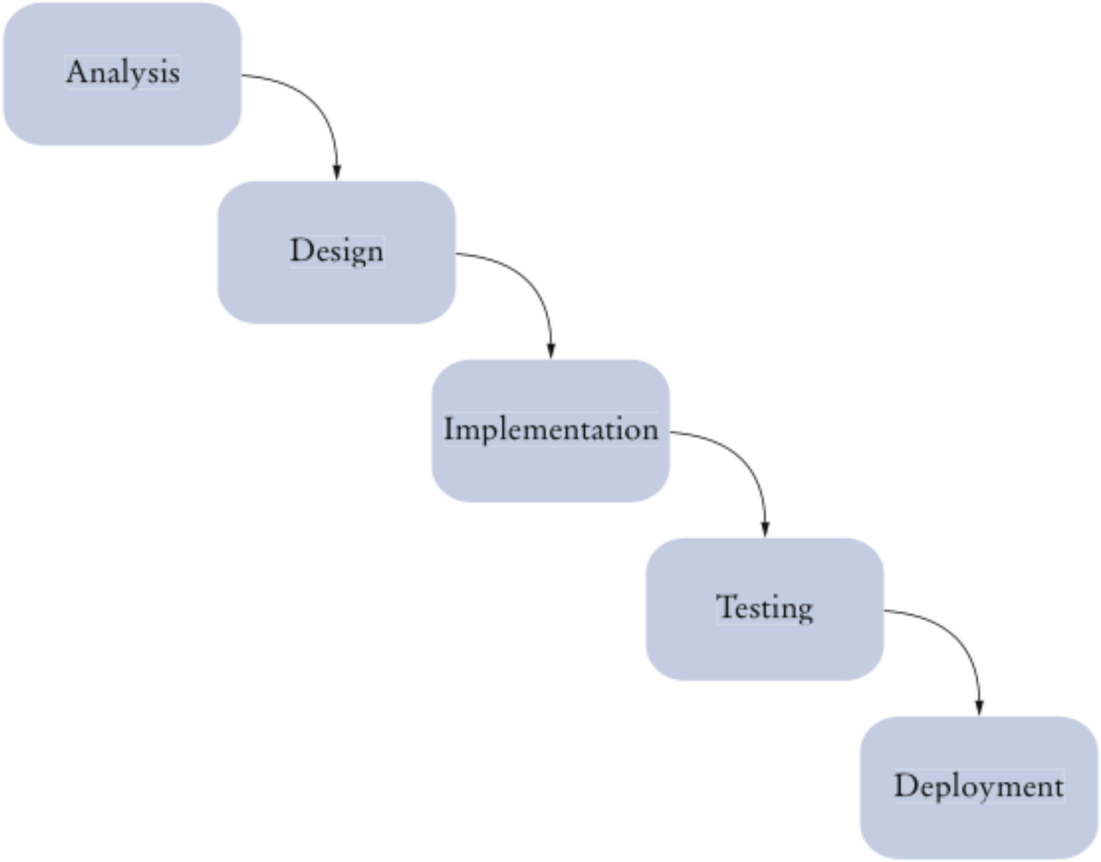


Figure 1 The Waterfall Model

Big Java by Cay Horstmann

The Spiral Model

- Breaks development process down into multiple phases
- Early phases focus on the construction of *prototypes*
- Lessons learned from development of one prototype can be applied to the next iteration

The Spiral Model

- Problem: Can lead to many iterations, and process can take too long to complete

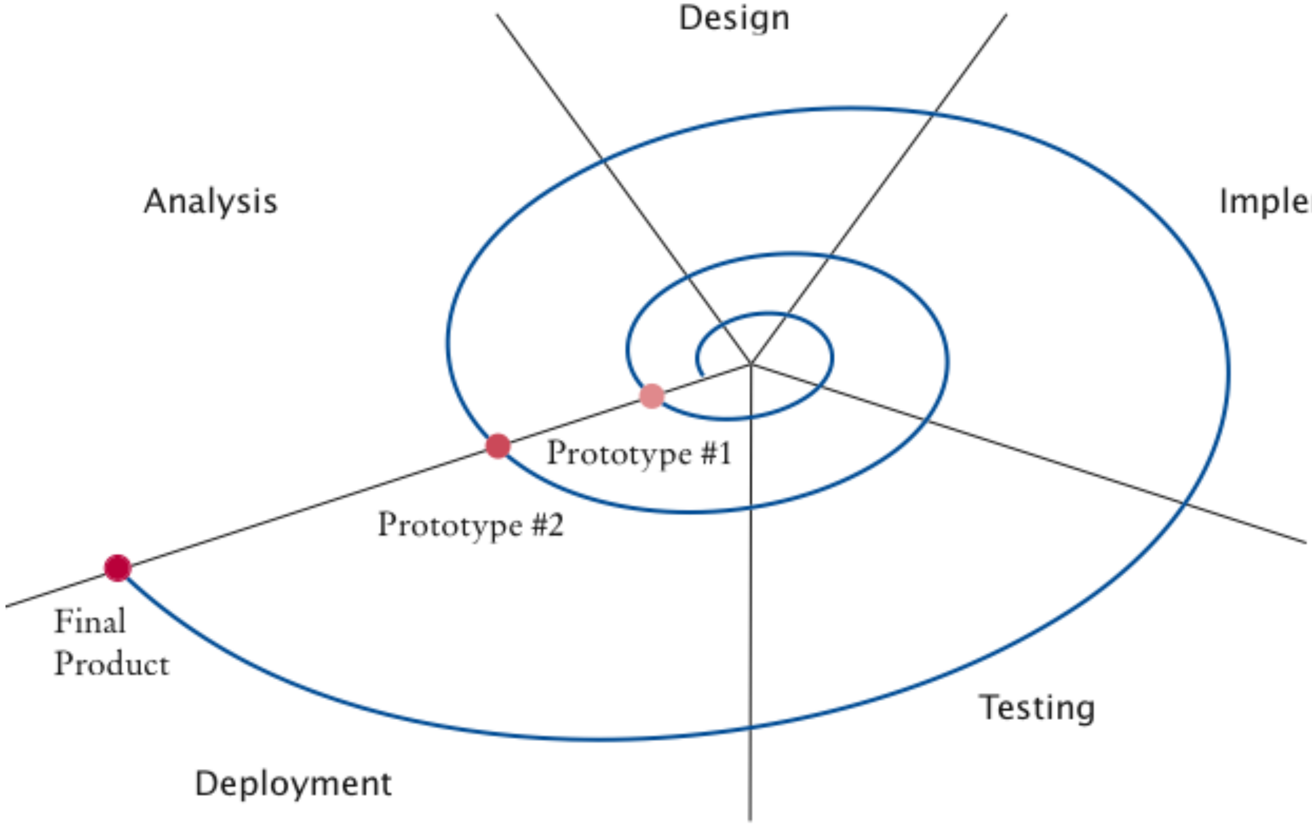


Figure 2 A Spiral Model

Big Java by Cay Horstmann

Activity Levels in the Rational Unified Process

Development process methodology by the inventors of UML

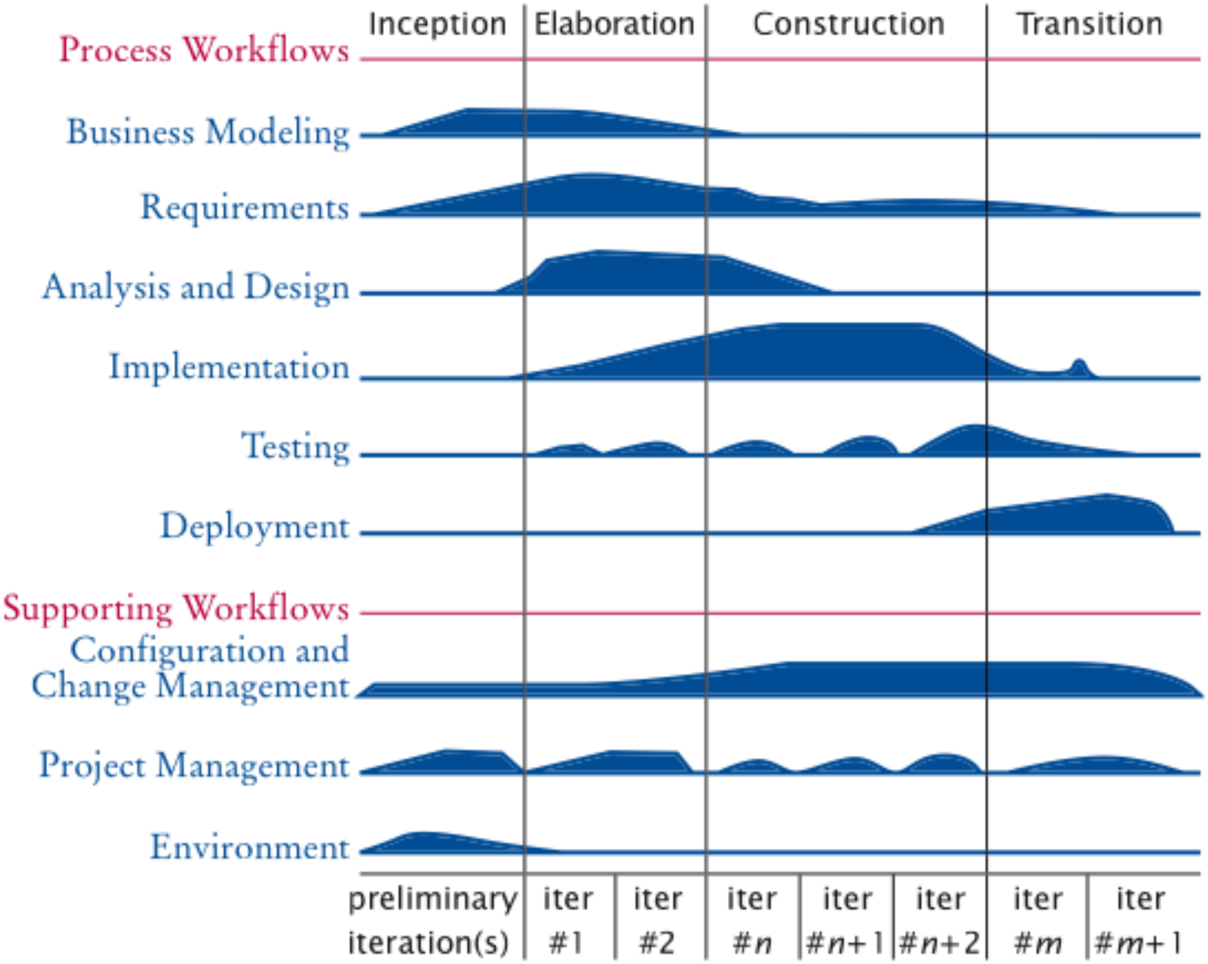


Figure 3 Activity Levels in the Rational Unified Process Methodology

Big Java by Cay Horstmann

Extreme Programming

- Strives for simplicity
- Removes formal structure
- Focuses on best practices

Extreme Programming

- Realistic planning
  - *Customers make business decisions*
  - *Programmers make technical decisions*
  - *Update plan when it conflicts with reality*
- Small releases
  - *Release a useful system quickly*
  - *Release updates on a very short cycle*
- Metaphor
  - *Programmers have a simple shared story that explains the system*

Extreme Programming

- Simplicity
  - *Design as simply as possible instead of preparing for future complexities*
- Testing
  - *Programmers and customers write test cases*
  - *Test continuously*
- Refactoring
  - *Restructure the system continuously to improve code and eliminate duplication*

Extreme Programming

- Pair programming
  - *Two programmers write code on the same computer*
- Collective ownership
  - *All programmers can change all code as needed*
- Continuous integration
  - *Build the entire system and test it whenever a task is complete*

Extreme Programming

- 40-hour week
  - *Don’t cover up unrealistic schedules with heroic effort*
- On-site customer
  - *A customer is accessible to the programming team at all times*
- Coding standards
  - *Follow standards that emphasize self-documenting code*

Self Check 12.1

Suppose you sign a contract, promising that you will, for an agreed-upon price, design, implement, and test a software package exactly as it has been specified in a requirements document. What is the primary risk you and your customer are facing with this business arrangement?

Self Check 12.2

Does Extreme Programming follow a waterfall or a spiral model?

Self Check 12.3

What is the purpose of the “on-site customer” in Extreme Programming?

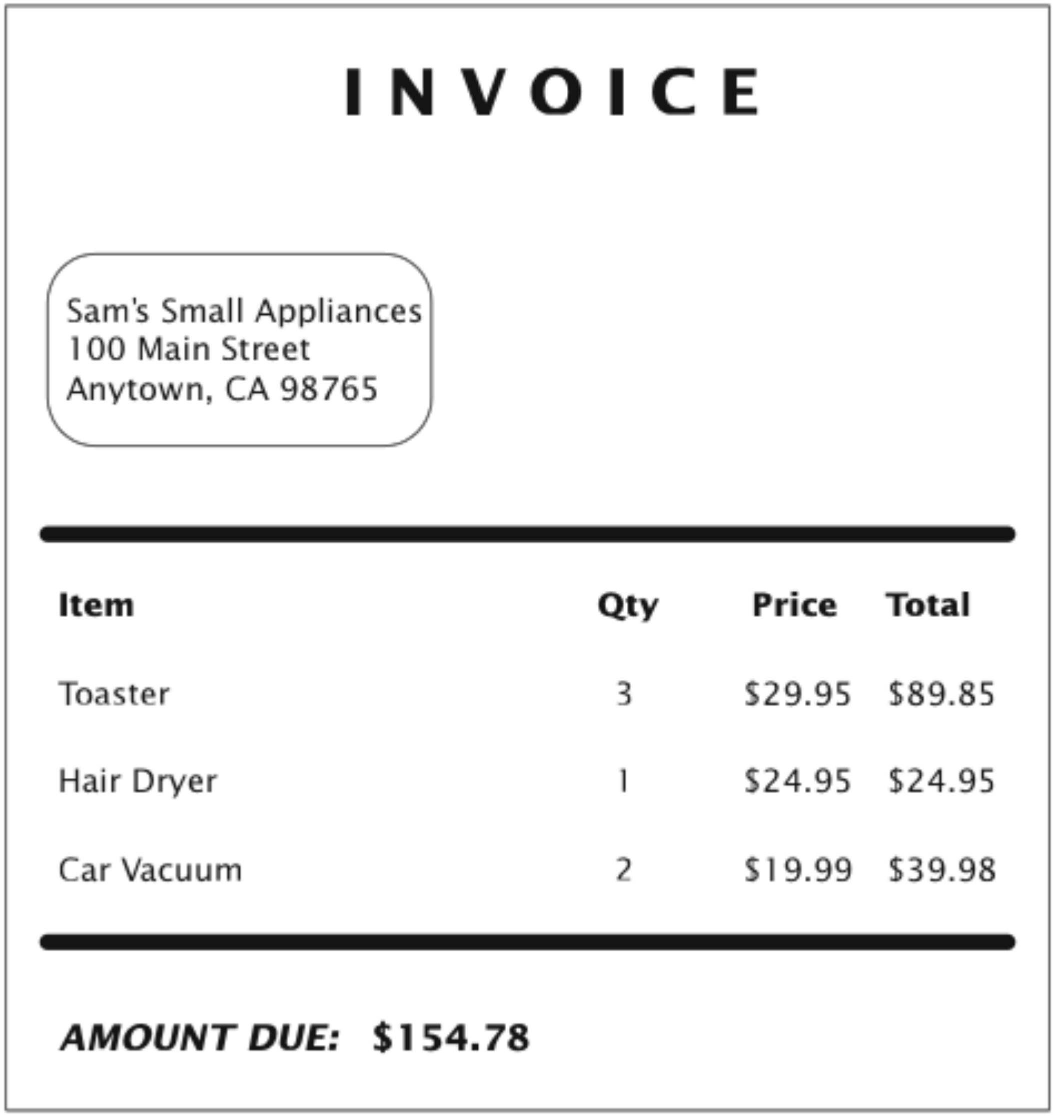
Object-Oriented Design

1. Discover classes
2. Determine responsibilities of each class
3. Describe relationships between the classes

Discovering Classes

- A class represents some useful concept
- Concrete entities: Bank accounts, ellipses, and products
- Abstract concepts: Streams and windows
- Find classes by looking for nouns in the task description
- Define the behavior for each class
- 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 4
An Invoice

Example: Invoice

- Classes that come to mind: `Invoice`, `LineItem`, and `Customer`
- Good idea to keep a list of candidate classes
- Brainstorm, simply put all ideas for classes onto the list
- You can cross not useful ones later

Finding Classes

- Keep the following points in mind:
  - *Class represents set of objects with the same behavior*
    - *Entities with multiple occurrences in problem description are good candidates for objects*
    - *Find out what they have in common*
    - *Design classes to capture commonalities*
  - *Represent some entities as objects, others as primitive types*
    - *Should we make a class* `Address` *or use a* `String`*?*
  - *Not all classes can be discovered in analysis phase*
  - *Some classes may already exist*

CRC Card

- Describes a **c**lass, its **r**esponsibilities, and its **c**ollaborators
- 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

Continued

CRC Card

- Indicate what other classes are needed to fulfill responsibility (collaborators)

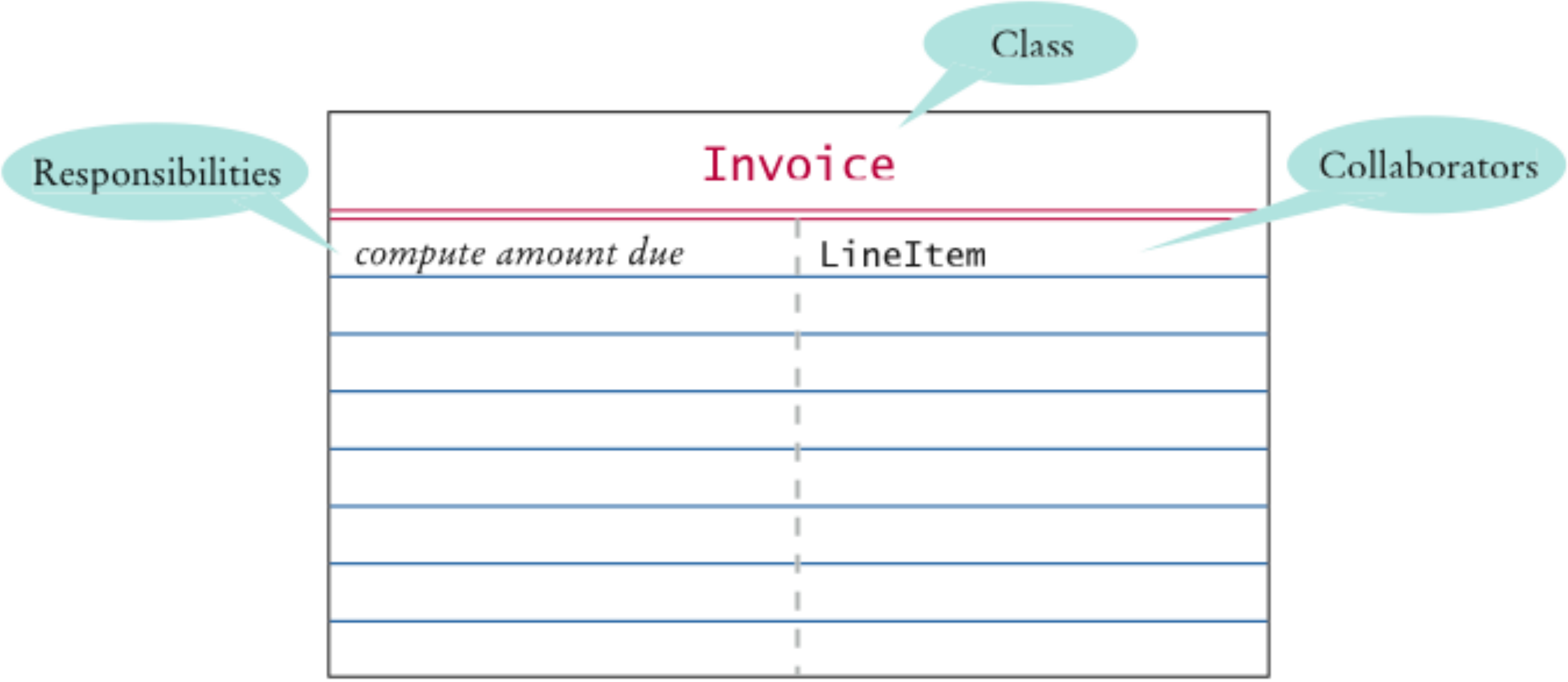


Figure 5 A CRC Card

Big Java by Cay Horstmann

Self Check 12.4

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

Self Check 12.5

Looking at the invoice in Figure 4, what is a likely responsibility of the `Customer` class?

Self Check 12.6

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

Relationships Between Classes

- Inheritance
- Aggregation
- Dependency

Inheritance

- *Is-a* relationship
- Relationship between a more general class (superclass) and a more specialized class (subclass)
- Every savings account is a bank account
- Every circle is an ellipse (with equal width and height)
- It is sometimes abused
    - *Should the class `Tire` be a subclass of a class `Circle`?*
        - *The has-a relationship would be more appropriate*

Aggregation

- *Has-a* relationship
- Objects of one class contain references to objects of another class
- Use an instance variable
    - *A tire has a circle as its boundary:*

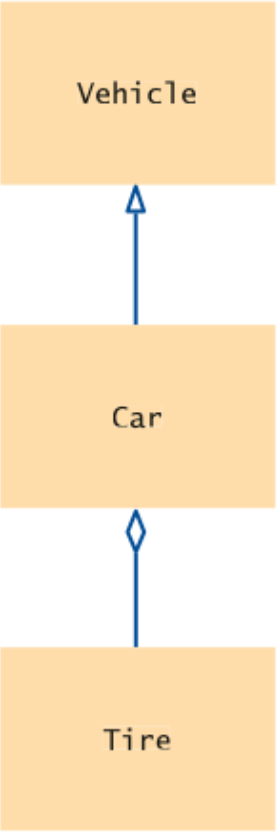
```
class Tire
{
   ...
   private String rating;
   private Circle boundary;
}
```

- Every car has a tire (in fact, it has four)

Example

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

Figure 6
UML Notation for
Inheritance and Aggregation



Dependency

- *Uses* relationship
- Example: Many of our applications depend on the `Scanner` class to read input
- Aggregation is a stronger form of dependency
- Use aggregation to remember another object between method calls

UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance	────▷	Solid	Triangle
Interface Implementation	- - - - - - - -▷	Dotted	Triangle
Aggregation	◇────	Solid	Diamond
Dependency	- - - - - - - - ->	Dotted	Open

Self Check 12.7

Consider the `Bank` and `BankAccount` classes of Chapter 7. How are they related?

Self Check 12.8

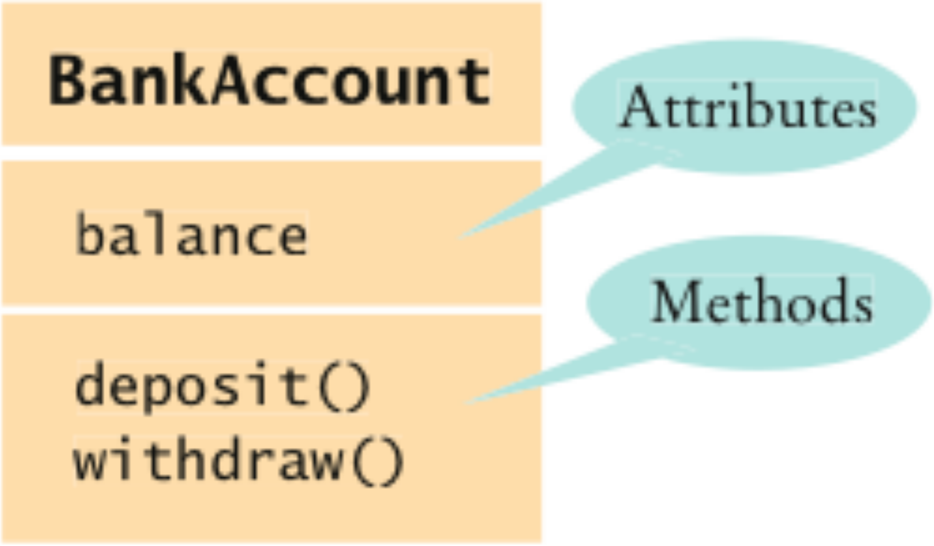
Consider the `BankAccount` and `SavingsAccount` objects of Chapter 10. How are they related?

Self Check 12.9

Consider the `BankAccountTester` class of Chapter 3. Which classes does it depend on?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

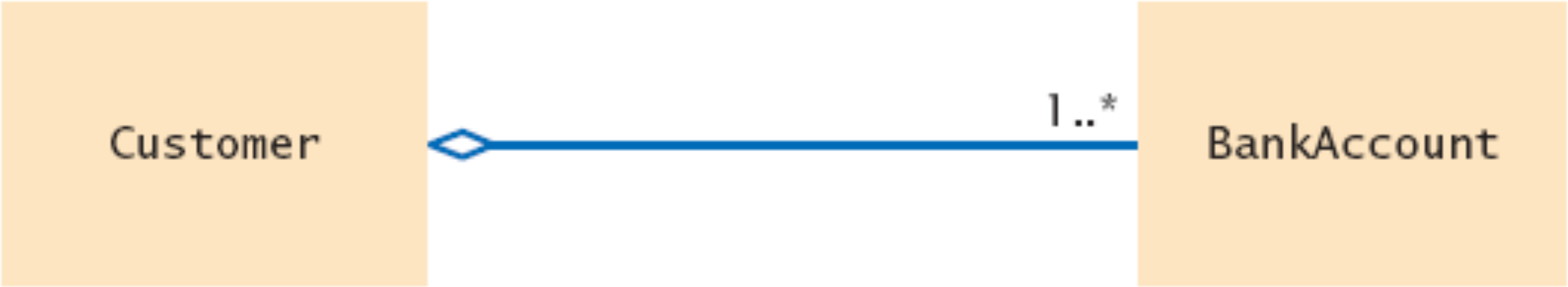
Attributes and Methods in UML Diagrams



Attributes and Methods in a Class Diagram

Multiplicities

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

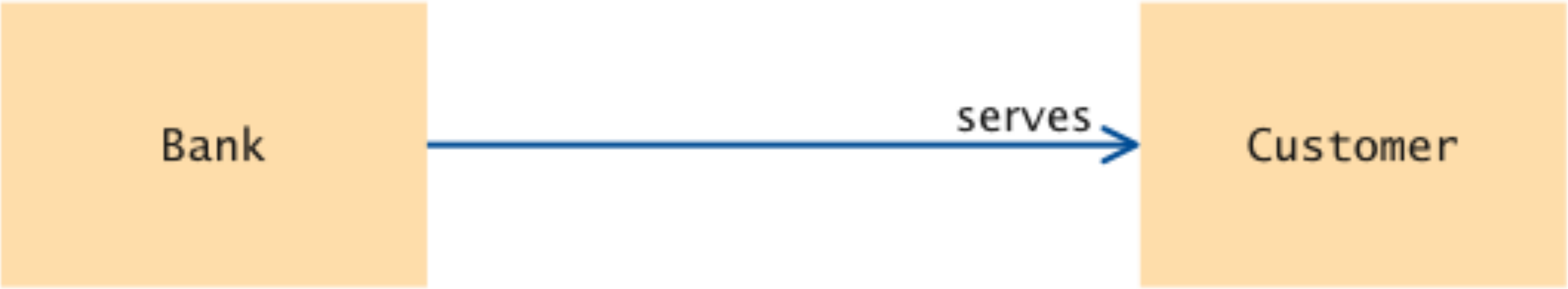


An Aggregation Relationship with Multiplicities

Big Java by Cay Horstmann

Aggregation and Association

- 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

Big Java by Cay Horstmann

Five-Part 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

Case Study: Printing an Invoice — 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

Case Study: Sample Invoice

INVOICE

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

Case Study: Printing an Invoice — CRC Cards

- Discover classes
- Nouns are possible classes:

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

Case Study: Printing an Invoice — CRC Cards

- Analyze classes:

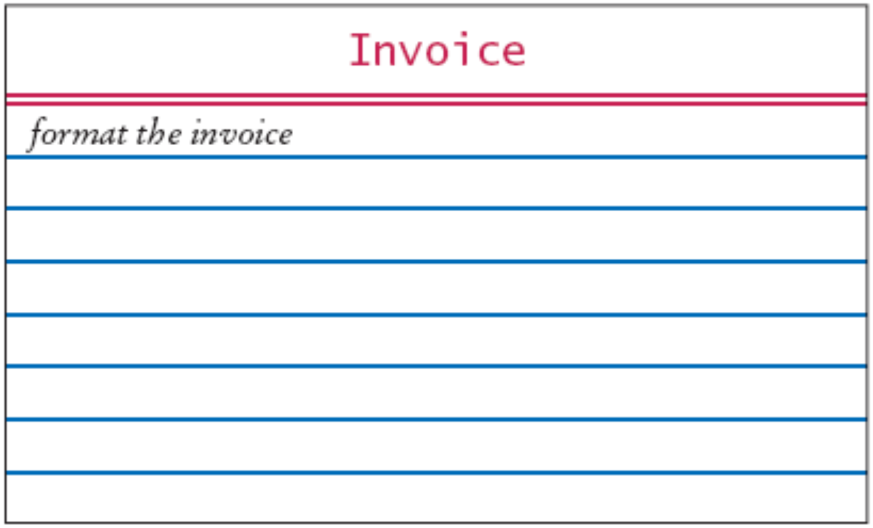
```
Invoice
Address
LineItem     // Records the product and the quantity
Product
Description  // variable of the Product class
Price        // variable of the Product class
Quantity     // Not an attribute of a Product
Total        // Computed - not stored anywhere
Amount Due   // Computed - not stored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

- Classes after a process of elimination:

```
Invoice
Address
LineItem
Product
```

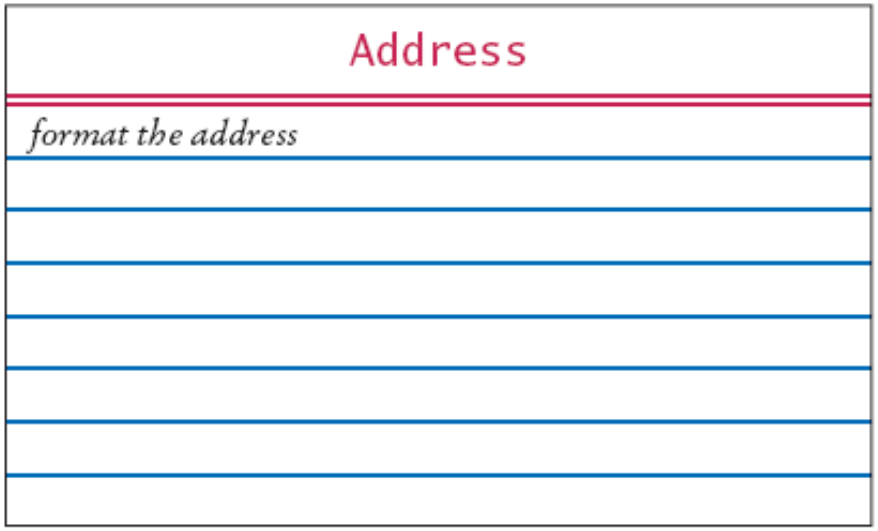
CRC Cards for Printing Invoice

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



Invoice

format the invoice

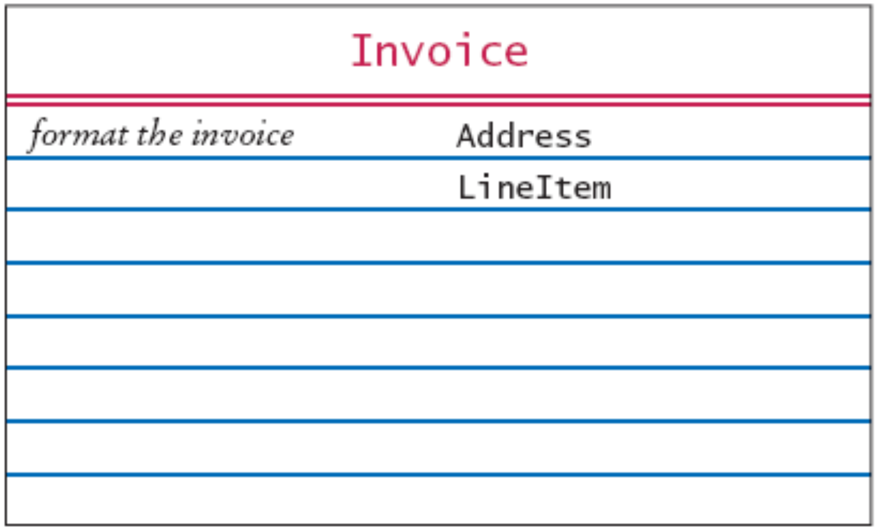


Address

format the address

CRC Cards for Printing Invoice

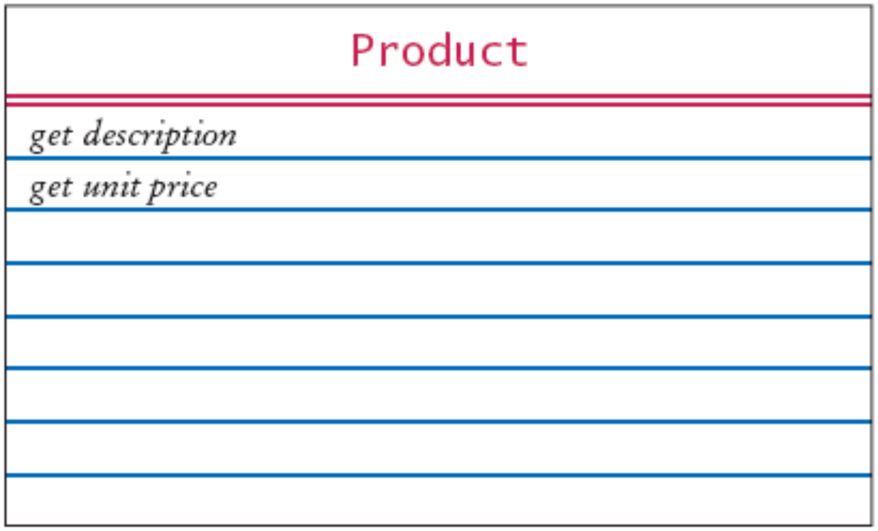
Add collaborators to invoice card:



Invoice	
format the invoice	Address
	LineItem

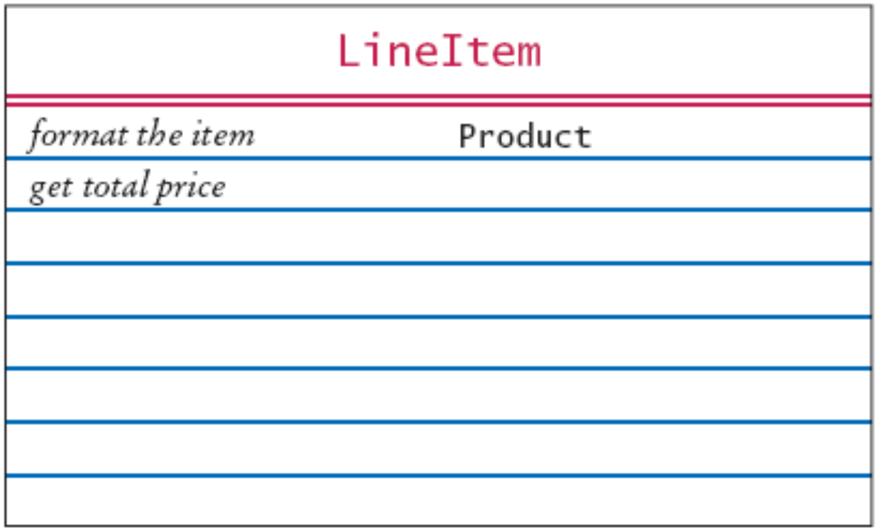
CRC Cards for Printing Invoice

`Product` and `LineItem` CRC cards:



Product

get description	
get unit price	

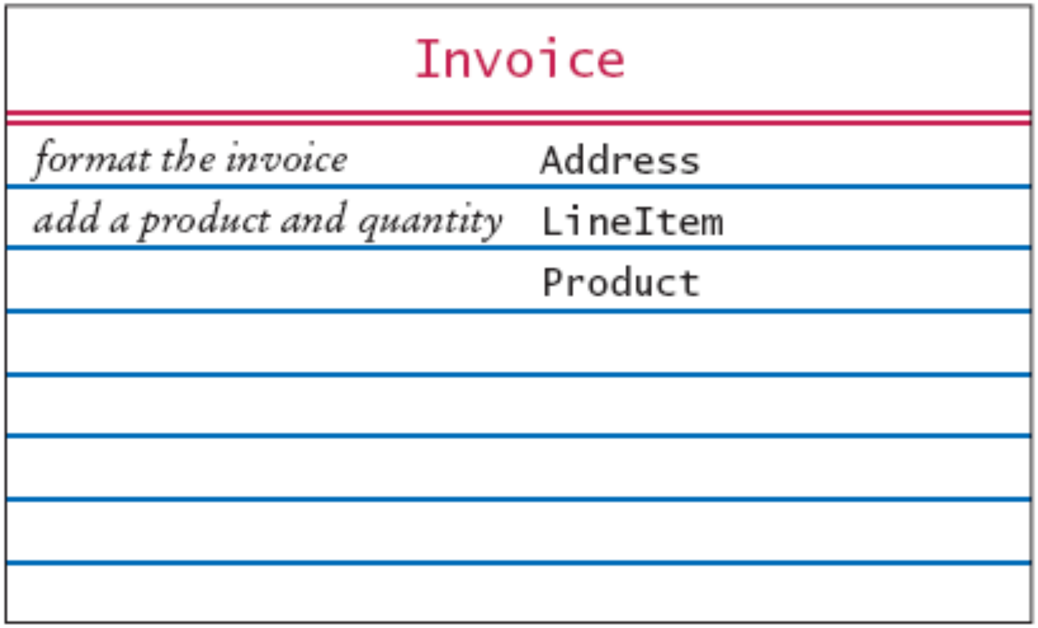


LineItem

format the item	Product
get total 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

Printing an Invoice — UML Diagrams

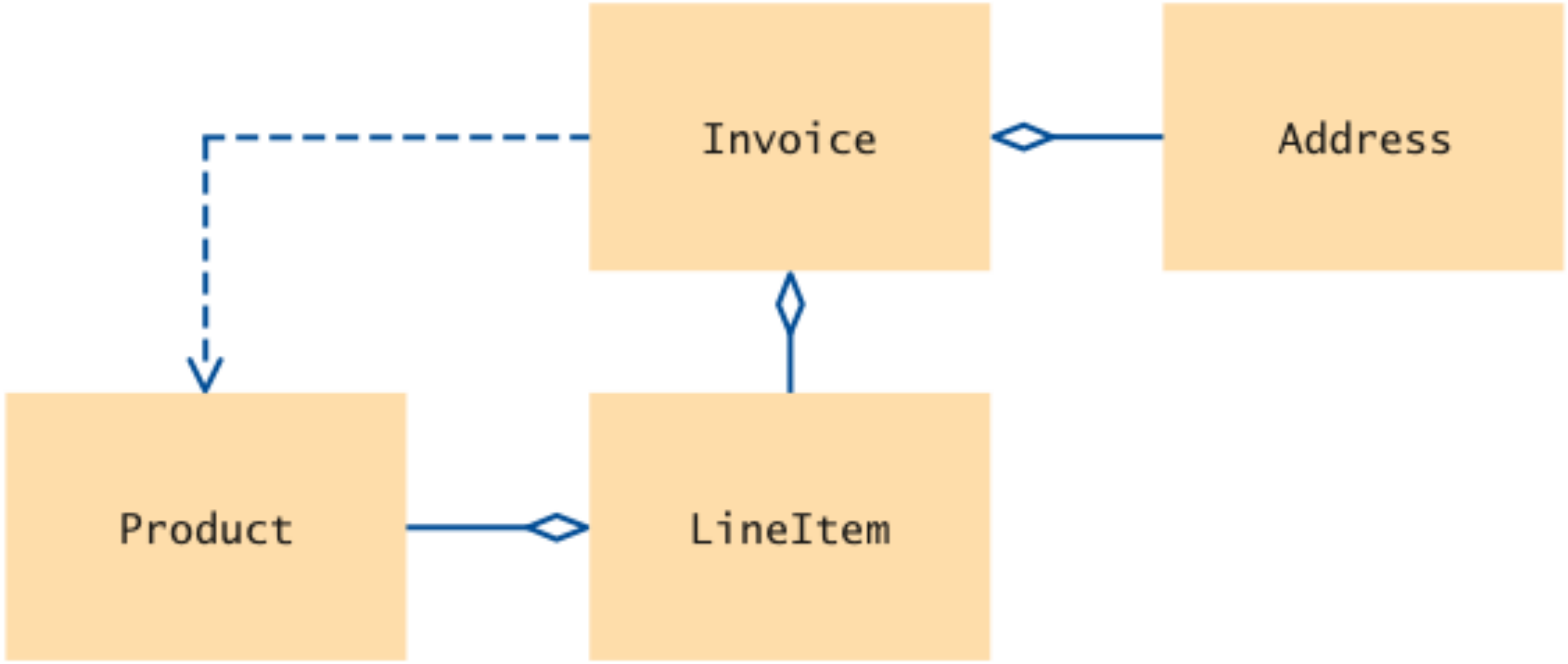


Figure 7 The Relationships Between the Invoice Classes

Printing an Invoice — Method Documentation

- Use `javadoc` documentation to record the behavior of the classes
- 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)
   {
   }
```

Continued

Method Documentation — `Invoice` Class (cont.)

```
   /**
      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()
   {
   }
```

Continued

Method Documentation — `LineItem` Class (cont.)

```
   /**
      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()
   {
   }
```

Continued

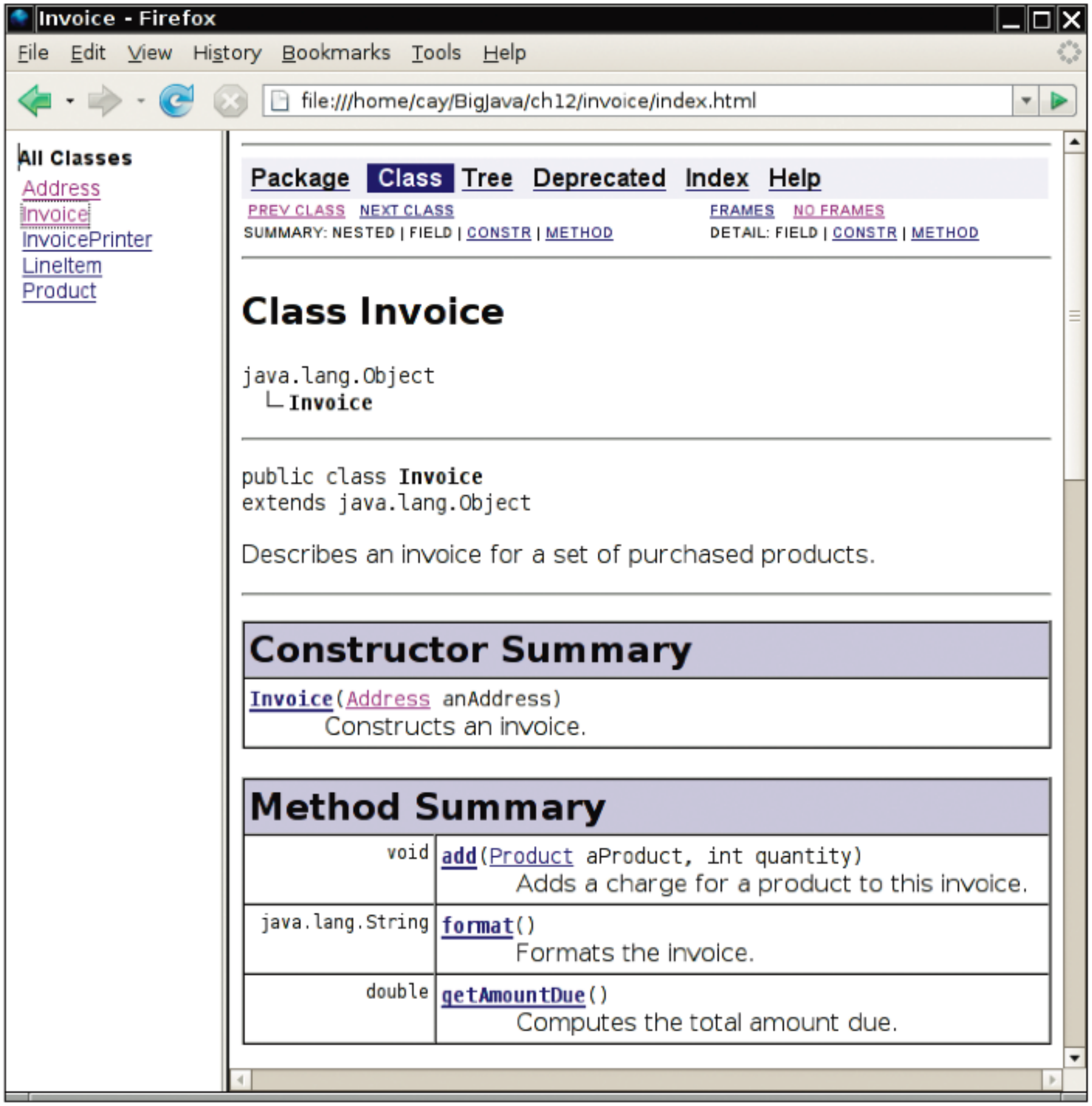
Method Documentation — `Product` Class (cont.)

```
   /**
      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 - 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.

Figure 8 The Class Documentation in HTML Format

Big Java by Cay Horstmann

Printing an Invoice — Implementation

- The UML diagram will give instance variables
- Look for associated classes
  - *They yield instance variables*

Big Java by Cay Horstmann

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;
}
```

ch12/invoice/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
            = new Address("Sam&apos;s Small Appliances",
               "100 Main Street", "Anytown", "CA", "98765");

      Invoice samsInvoice = new Invoice(samsAddress);
      samsInvoice.add(new Product("Toaster", 29.95), 3);
      samsInvoice.add(new Product("Hair dryer", 24.95), 1);
      samsInvoice.add(new Product("Car vacuum", 19.99), 2);

      System.out.println(samsInvoice.format());
   }
}
```

ch12/invoice/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;

   /**
      Constructs an invoice.
      @param anAddress the billing address
   */
   public Invoice(Address anAddress)
   {
      items = new ArrayList<LineItem>();
      billingAddress = anAddress;
   }

```

Continued

ch12/invoice/Invoice.java (cont.)

```
   /**
      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)
   {
      LineItem anItem = new LineItem(aProduct, quantity);
      items.add(anItem);
   }

```

Continued

ch12/invoice/Invoice.java (cont.)

```
   /**
      Formats the invoice.
      @return the formatted invoice
   */
   public String format()
   {
      String r =  "                     I N V O I C E\n\n"
            + billingAddress.format()
            + String.format("\n\n%-30s%8s%5s%8s\n",
               "Description", "Price", "Qty", "Total");

      for (LineItem item : items)
      {
         r = r + item.format() + "\n";
      }

      r = r + String.format("\nAMOUNT DUE: $%8.2f", getAmountDue());

      return r;
   }

```

Continued

ch12/invoice/Invoice.java (cont.)

```java
   /**
      Computes the total amount due.
      @return the amount due
   */
   public double getAmountDue()
   {
      double amountDue = 0;
      for (LineItem item : items)
      {
         amountDue = amountDue + item.getTotalPrice();
      }
      return amountDue;
   }
}
```

ch12/invoice/LineItem.java

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

   /**
      Constructs an item from the product and quantity.
      @param aProduct the product
      @param aQuantity the item quantity
   */
   public LineItem(Product aProduct, int aQuantity)
   {
      theProduct = aProduct;
      quantity = aQuantity;
   }

```

Continued

ch12/invoice/LineItem.java (cont.)

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

   /**
      Formats this item.
      @return a formatted string of this item
   */
   public String format()
   {
      return String.format("%-30s%8.2f%5d%8.2f",
         theProduct.getDescription(), theProduct.getPrice(),
         quantity, getTotalPrice());
   }
}
```

ch12/invoice/Product.java

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

   /**
      Constructs a product from a description and a price.
      @param aDescription the product description
      @param aPrice the product price
   */
   public Product(String aDescription, double aPrice)
   {
      description = aDescription;
      price = aPrice;
   }

```

Continued

ch12/invoice/Product.java (cont.)

```java
    /**
        Gets the product description.
        @return the description
    */
    public String getDescription()
    {
        return description;
    }

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

```

ch12/invoice/Address.java

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

   /**
      Constructs a mailing address.
      @param aName the recipient name
      @param aStreet the street
      @param aCity the city
      @param aState the two-letter state code
      @param aZip the ZIP postal code
   */
```

Continued

ch12/invoice/Address.java (cont.)

```java
   public Address(String aName, String aStreet,
         String aCity, String aState, String aZip)
   {
      name = aName;
      street = aStreet;
      city = aCity;
      state = aState;
      zip = aZip;
   }

   /**
      Formats the address.
      @return the address as a string with three lines
   */
   public String format()
   {
      return name + "\n" + street + "\n"
            + city + ", " + state + " " + zip;
   }
}

```

Self Check 12.10

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

Self Check 12.11

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

Case Study: An Automatic Teller Machine — Requirements

- ATM is used by bank customers. A customer has a
  - *Checking account*
  - *Savings account*
  - *Customer number*
  - *PIN*

Case Study: An Automatic Teller Machine — Requirements

- Customers can select an account
- The balance of the selected account is displayed
- Then, customer can deposit and withdraw money
- Process is repeated until the customer chooses to exit

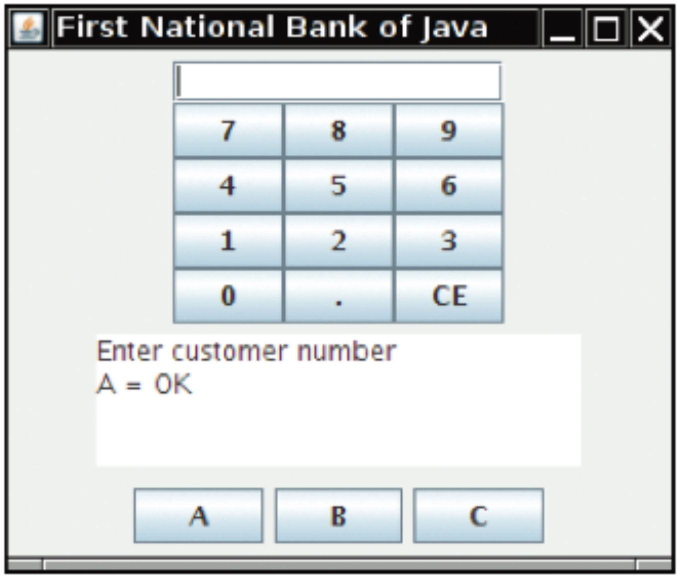
Case Study: An Automatic Teller Machine — Requirements

- Two separate interfaces:
    - *GUI that closely mimics an actual ATM*
    - *Text-based interface*

Case Study: An Automatic Teller Machine — Requirements

- GUI Interface
  - *Keypad*
  - *Display*
  - *Buttons A, B, C*
  - *Button functions depend on the state of the machine*

Figure 9
Graphical User Interface
for the Automatic Teller Machine



Enter customer number
A = OK

Case Study: An Automatic Teller Machine — Requirements

- At start up the customer is expected to
  - *Enter customer number*
  - *Press the A button*
  - *The display shows:*

```
Enter Customer Number
A = OK
```

Case Study: An Automatic Teller Machine — Requirements

- The customer is expected to
  - *Enter a PIN*
  - *Press A button*
  - *The display shows:*

```
Enter PIN
A = OK
```

Case Study: An Automatic Teller Machine — Requirements

- Search for the customer number and PIN
  - *If it matches a bank customer, proceed*
  - *Else return to start up screen*

Case Study: An Automatic Teller Machine — Requirements

- If the customer is authorized
  - *The display shows:*

```
Select Account
A = Checking
B = Savings
C = Exit
```

Case Study: An Automatic Teller Machine — Requirements

- If the user presses C
  - *The ATM reverts to its original state*
  - *ATM asks next user to enter a customer number*
- If the user presses A or B
  - *The ATM remembers selected account*
  - *The display shows:*

```
Balance = balance of selected account
Enter amount and select transaction
A = Withdraw
B = Deposit
C = Cancel
```

Case Study: An Automatic Teller Machine — Requirements

- If the user presses A or B
  - *The value entered is withdrawn or deposited*
  - *Simulation: No money is dispensed and no deposit is accepted*
  - *The ATM reverts to previous state*
- If the user presses C
  - *The ATM reverts to previous state*

Case Study: An Automatic Teller Machine — Requirements

- Text-based interaction
  - *Read input from* `System.in` *instead of the buttons*
  - *Here is a typical dialog:*

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```

An Automatic Teller Machine – CRC Cards

Nouns are possible classes:

```
ATM
User
Keypad
Display
Display message
Button
State
Bank account
Checking account
Savings account
Customer
Customer number
PIN
Bank
```

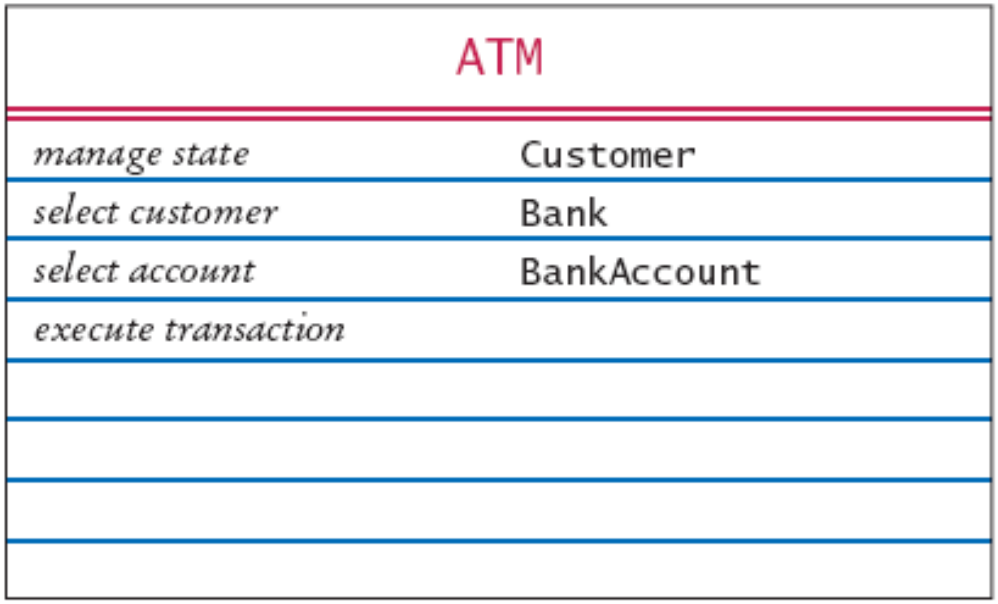
Big Java by Cay Horstmann

CRC Cards for Automatic Teller Machine

Customer	
get accounts	
match number and PIN	

Bank	
find customer	Customer
read customers	

CRC Cards for Automatic Teller Machine



ATM

manage state	Customer
select customer	Bank
select account	BankAccount
execute transaction	

ATM States

1. START: Enter customer ID
2. PIN: Enter PIN
3. ACCOUNT: Select account
4. TRANSACT: Select transaction

Big Java by Cay Horstmann

State Diagram for ATM Class

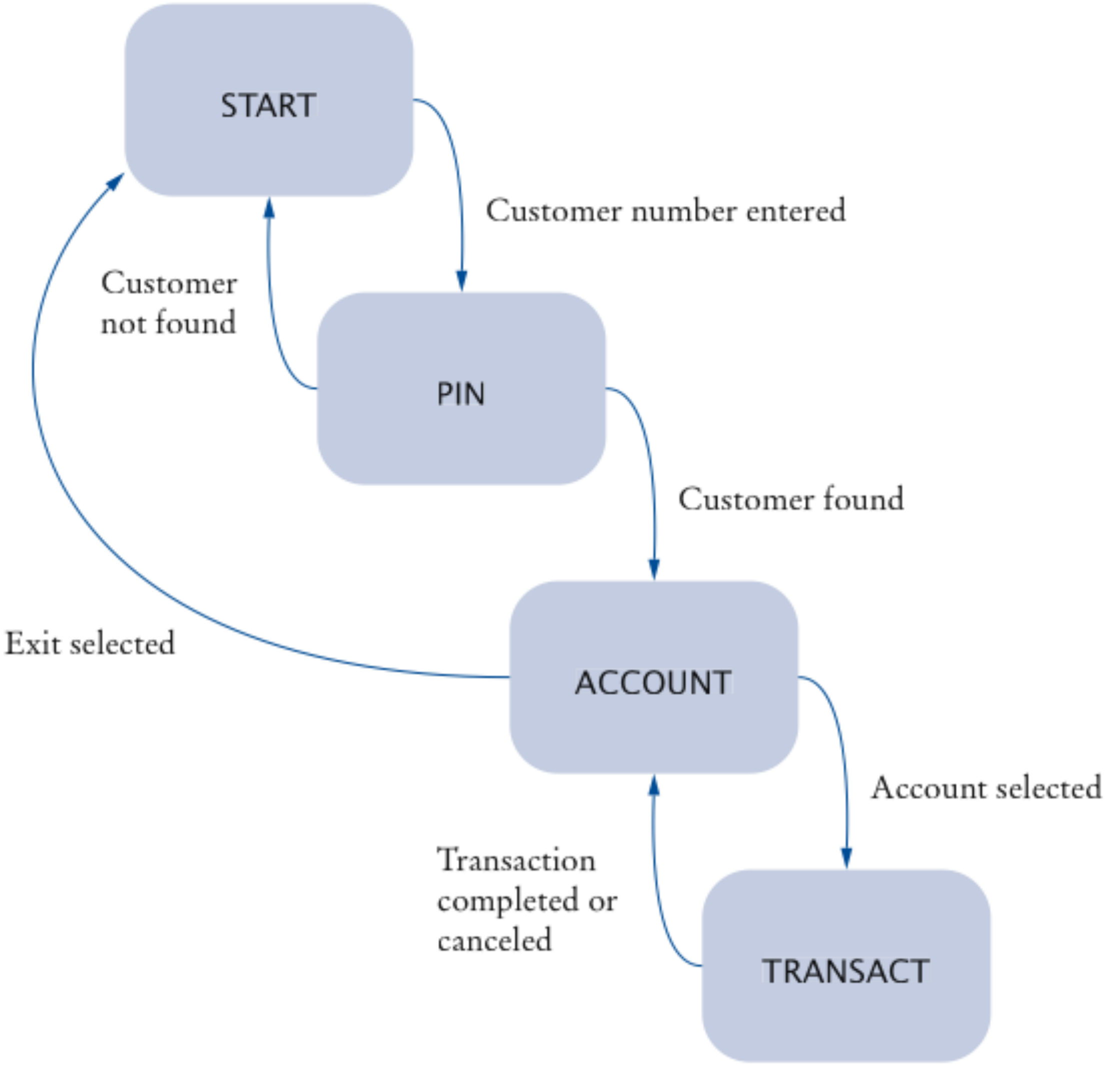


Figure 10 State Diagram for the ATM Class

Big Java by Cay Horstmann

An Automatic Teller Machine – UML Diagrams

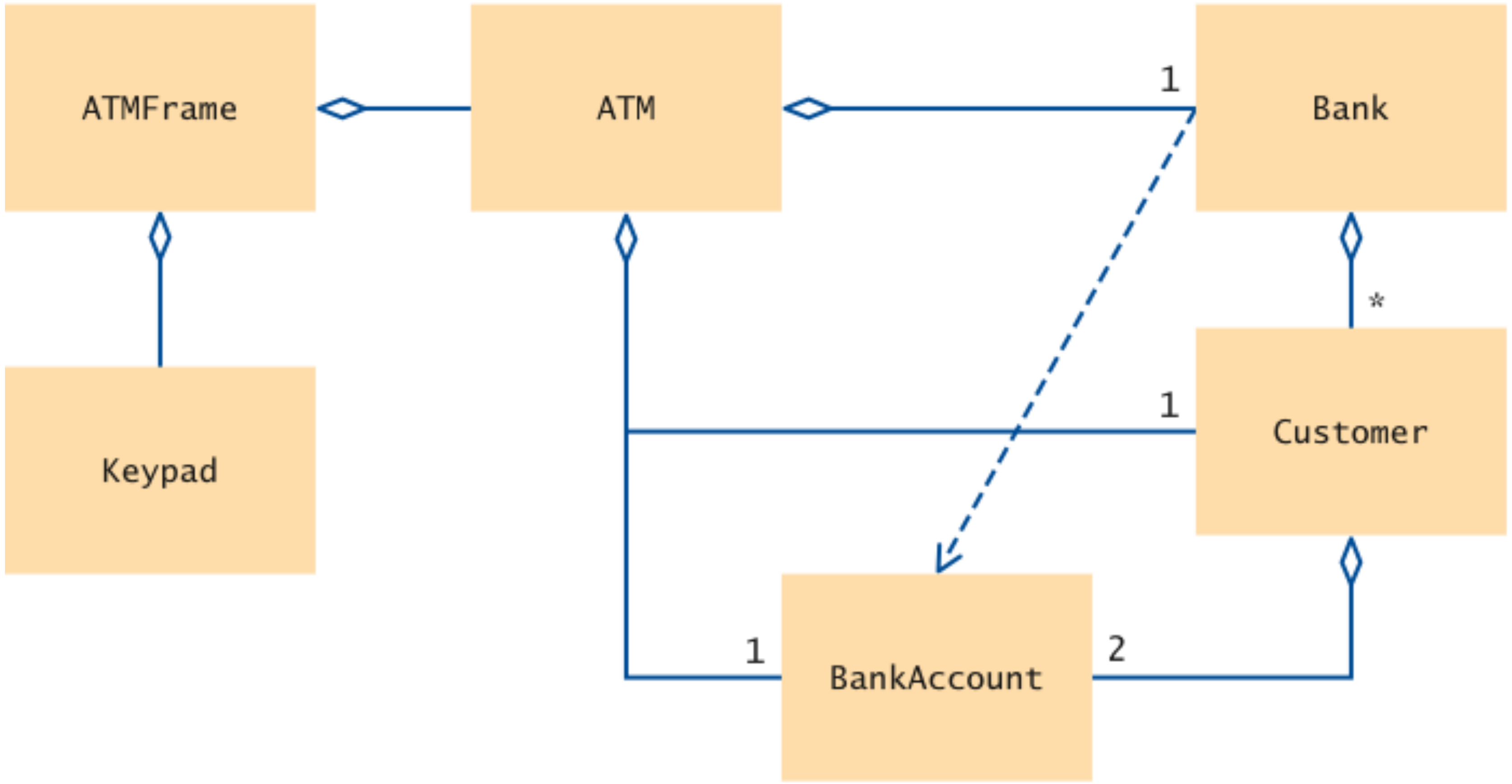


Figure 11 Relationships Between the ATM Classes

Big Java by Cay Horstmann

Method Documentation ATM Class

```
/**
   An ATM that accesses a bank.
*/
public class ATM
{
   /**
      Constructs an ATM for a given bank.
      @param aBank the bank to which this ATM connects
   */
   public ATM(Bank aBank) { }
   /**
      Sets the current customer number
      and sets state to PIN.
      (Precondition: state is START)
      @param number the customer number
   */
   public void setCustomerNumber(int number) { }
```

Continued

Method Documentation ATM Class (cont.)

```
/**
   Finds customer in bank.
   If found sets state to ACCOUNT, else to START.
   (Precondition: state is PIN)
   @param pin the PIN of the current customer
*/
public void selectCustomer(int pin) { }
/**
   Sets current account to checking or savings. Sets
   state to TRANSACT.
   (Precondition: state is ACCOUNT or TRANSACT)
   @param account one of CHECKING or SAVINGS
*/
```

Continued

Method Documentation ATM Class (cont.)

```
   public void selectAccount(int account) { }
   /**
      Withdraws amount from current account.
      (Precondition: state is TRANSACT)
      @param value the amount to withdraw
   */
   public void withdraw(double value) { }
   ...
}
```

An Automatic Teller Machine — Implementation

- Start implementation with classes that don't depend on others
    - `Keypad`
    - `BankAccount`
- Then implement `Customer` which depends only on `BankAccount`
- This bottom-up approach allows you to test your classes individually

An Automatic Teller Machine — Implementation

- Aggregated classes in UML diagram give instance variables

```
public class ATM
{
   private Bank theBank;
   . . .
}
```

- From description of ATM states, it is clear that we require additional instance variables:

```
public class ATM
{
   private int state;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   . . .
}
```

An Automatic Teller Machine — Implementation

- Most methods are very straightforward to implement
- Consider `selectCustomer`:

```
/**
   Finds customer in bank.
   If found sets state to ACCOUNT, else to START.
   (Precondition: state is PIN)
   @param pin the PIN of the current customer
*/
```

Continued

An Automatic Teller Machine — Implementation (cont.)

- Description can be almost literally translated to Java instructions:

```
public void selectCustomer(int pin)
{
   assert state == PIN;
   currentCustomer = theBank.findCustomer(customerNumber,
         pin);
   if (currentCustomer == null)
      state = START;
   else
      state = ACCOUNT;
}
```

ch12/atm/ATM.java

```java
/**
   An ATM that accesses a bank.
*/
public class ATM
{
   public static final int CHECKING = 1;
   public static final int SAVINGS = 2;

   private int state;
   private int customerNumber;
   private Customer currentCustomer;
   private BankAccount currentAccount;
   private Bank theBank;

   public static final int START = 1;
   public static final int PIN = 2;
   public static final int ACCOUNT = 3;
   public static final int TRANSACT = 4;

```

Continued

ch12/atm/ATM.java (cont.)

```java
   /**
      Constructs an ATM for a given bank.
      @param aBank the bank to which this ATM connects
   */
   public ATM(Bank aBank)
   {
      theBank = aBank;
      reset();
   }

   /**
      Resets the ATM to the initial state.
   */
   public void reset()
   {
      customerNumber = -1;
      currentAccount = null;
      state = START;
   }

```

Continued

ch12/atm/ATM.java (cont.)

```
    /**
       Sets the current customer number
       and sets state to PIN.
       (Precondition: state is START)
       @param number the customer number.
    */
    public void setCustomerNumber(int number)
    {
       assert state == START;
       customerNumber = number;
       state = PIN;
    }

```

Continued

ch12/atm/ATM.java (cont.)

```
   /**
      Finds customer in bank.
      If found sets state to ACCOUNT, else to START.
      (Precondition: state is PIN)
      @param pin the PIN of the current customer
   */
   public void selectCustomer(int pin)
   {
      assert state == PIN;
      currentCustomer = theBank.findCustomer(customerNumber, pin);
      if (currentCustomer == null)
         state = START;
      else
         state = ACCOUNT;
   }

```

Continued

ch12/atm/ATM.java (cont.)

```
   /**
      Sets current account to checking or savings. Sets
      state to TRANSACT.
      (Precondition: state is ACCOUNT or TRANSACT)
      @param account one of CHECKING or SAVINGS
   */
   public void selectAccount(int account)
   {
      assert state == ACCOUNT || state == TRANSACT;
      if (account == CHECKING)
         currentAccount = currentCustomer.getCheckingAccount();
      else
         currentAccount = currentCustomer.getSavingsAccount();
      state = TRANSACT;
   }

```

Continued

ch12/atm/ATM.java (cont.)

```
   /**
      Withdraws amount from current account.
      (Precondition: state is TRANSACT)
      @param value the amount to withdraw
   */
   public void withdraw(double value)
   {
      assert state == TRANSACT;
      currentAccount.withdraw(value);
   }

   /**
      Deposits amount to current account.
      (Precondition: state is TRANSACT)
      @param value the amount to deposit
   */
   public void deposit(double value)
   {
      assert state == TRANSACT;
      currentAccount.deposit(value);
   }

```

Continued

ch12/atm/ATM.java (cont.)

```java
   /**
      Gets the balance of the current account.
      (Precondition: state is TRANSACT)
      @return the balance
   */
   public double getBalance()
   {
      assert state == TRANSACT;
      return currentAccount.getBalance();
   }

   /**
      Moves back to the previous state.
   */
   public void back()
   {
      if (state == TRANSACT)
         state = ACCOUNT;
      else if (state == ACCOUNT)
         state = PIN;
      else if (state == PIN)
         state = START;
   }

```

Continued

ch12/atm/ATM.java (cont.)

```java
   /**
      Gets the current state of this ATM.
      @return the current state
   */
   public int getState()
   {
      return state;
   }
}
```

ch12/atm/Bank.java

```java
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Scanner;

/**
   A bank contains customers with bank accounts.
*/
public class Bank
{
   private ArrayList<Customer> customers;

   /**
      Constructs a bank with no customers.
   */
   public Bank()
   {
      customers = new ArrayList<Customer>();
   }

```

Continued

ch12/atm/Bank.java (cont.)

```
   /**
      Reads the customer numbers and pins
      and initializes the bank accounts.
      @param filename the name of the customer file
   */
   public void readCustomers(String filename)
         throws IOException
   {
      Scanner in = new Scanner(new File(filename));
      while (in.hasNext())
      {
         int number = in.nextInt();
         int pin = in.nextInt();
         Customer c = new Customer(number, pin);
         addCustomer(c);
      }
      in.close();
   }

```

Continued

ch12/atm/Bank.java (cont.)

```java
   /**
      Adds a customer to the bank.
      @param c the customer to add
   */
   public void addCustomer(Customer c)
   {
      customers.add(c);
   }

   /**
      Finds a customer in the bank.
      @param aNumber a customer number
      @param aPin a personal identification number
      @return the matching customer, or null if no customer
      matches
   */
   public Customer findCustomer(int aNumber, int aPin)
   {
      for (Customer c : customers)
      {
         if (c.match(aNumber, aPin))
            return c;
      }
      return null;
   }
}
```

ch12/atm/Customer.java

```java
/**
   A bank customer with a checking and a savings account.
*/
public class Customer
{
   private int customerNumber;
   private int pin;
   private BankAccount checkingAccount;
   private BankAccount savingsAccount;

   /**
      Constructs a customer with a given number and PIN.
      @param aNumber the customer number
      @param aPin the personal identification number
   */
   public Customer(int aNumber, int aPin)
   {
      customerNumber = aNumber;
      pin = aPin;
      checkingAccount = new BankAccount();
      savingsAccount = new BankAccount();
   }

```

Continued

ch12/atm/Customer.java (cont.)

```
   /**
      Tests if this customer matches a customer number
      and PIN.
      @param aNumber a customer number
      @param aPin a personal identification number
      @return true if the customer number and PIN match
   */
   public boolean match(int aNumber, int aPin)
   {
      return customerNumber == aNumber && pin == aPin;
   }

   /**
      Gets the checking account of this customer.
      @return the checking account
   */
   public BankAccount getCheckingAccount()
   {
      return checkingAccount;
   }

```

Continued

ch12/atm/Customer.java (cont.)

```
   /**
      Gets the savings account of this customer.
      @return the checking account
   */
   public BankAccount getSavingsAccount()
   {
      return savingsAccount;
   }
}
```

ch12/atm/ATMSimulator.java

```java
import java.io.IOException;
import java.util.Scanner;

/**
   A text-based simulation of an automatic teller machine.
*/
public class ATMSimulator
{
   public static void main(String[] args)
   {
      ATM theATM;
      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch(IOException e)
      {
         System.out.println("Error opening accounts file.");
         return;
      }

```

Continued

ch12/atm/ATMSimulator.java (cont.)

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

      while (true)
      {
         int state = theATM.getState();
         if (state == ATM.START)
         {
            System.out.print("Enter customer number: ");
            int number = in.nextInt();
            theATM.setCustomerNumber(number);
         }
         else if (state == ATM.PIN)
         {
            System.out.print("Enter PIN: ");
            int pin = in.nextInt();
            theATM.selectCustomer(pin);
         }
```

Continued

ch12/atm/ATMSimulator.java (cont.)

```java
         else if (state == ATM.ACCOUNT)
         {
            System.out.print("A=Checking, B=Savings, C=Quit: ");
            String command = in.next();
            if (command.equalsIgnoreCase("A"))
               theATM.selectAccount(ATM.CHECKING);
            else if (command.equalsIgnoreCase("B"))
               theATM.selectAccount(ATM.SAVINGS);
            else if (command.equalsIgnoreCase("C"))
               theATM.reset();
            else
               System.out.println("Illegal input!");
         }
         else if (state == ATM.TRANSACT)
         {
            System.out.println("Balance=" + theATM.getBalance());
            System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
            String command = in.next();
```

Continued

ch12/atm/ATMSimulator.java (cont.)

```
            if (command.equalsIgnoreCase("A"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.deposit(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("B"))
            {
               System.out.print("Amount: ");
               double amount = in.nextDouble();
               theATM.withdraw(amount);
               theATM.back();
            }
            else if (command.equalsIgnoreCase("C"))
               theATM.back();
            else
               System.out.println("Illegal input!");
         }
      }
   }
}
```

Continued

ch12/atm/ATMSimulator.java (cont.)

Program Run:

```
Enter account number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
...
```

ch12/atm/ATMViewer.java

```java
import java.io.IOException;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/**
   A graphical simulation of an automatic teller machine.
*/
public class ATMViewer
{
   public static void main(String[] args)
   {
      ATM theATM;

      try
      {
         Bank theBank = new Bank();
         theBank.readCustomers("customers.txt");
         theATM = new ATM(theBank);
      }
      catch(IOException e)
      {
         JOptionPane.showMessageDialog(null, "Error opening accounts file.");
         return;
      }

```

Continued

ch12/atm/ATMViewer.java (cont.)

```java
      JFrame frame = new ATMFrame(theATM);
      frame.setTitle("First National Bank of Java");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

ch12/atm/ATMFrame.java

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;

/**
   A frame displaying the components of an ATM.
*/
public class ATMFrame extends JFrame
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 300;

   private JButton aButton;
   private JButton bButton;
   private JButton cButton;

   private KeyPad pad;
   private JTextArea display;
```

Continued

ch12/atm/ATMFrame.java (cont.)

```

    private ATM theATM;

    /**
        Constructs the user interface of the ATM frame.
    */
    public ATMFrame(ATM anATM)
    {
        theATM = anATM;

        // Construct components
        pad = new KeyPad();

        display = new JTextArea(4, 20);

        aButton = new JButton("  A  ");
        aButton.addActionListener(new AButtonListener());

        bButton = new JButton("  B  ");
        bButton.addActionListener(new BButtonListener());

```

Continued

ch12/atm/ATMFrame.java (cont.)

```
      cButton = new JButton("   C   ");
      cButton.addActionListener(new CButtonListener());

      // Add components

      JPanel buttonPanel = new JPanel();
      buttonPanel.add(aButton);
      buttonPanel.add(bButton);
      buttonPanel.add(cButton);

      setLayout(new FlowLayout());
      add(pad);
      add(display);
      add(buttonPanel);
      showState();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

```

Continued

ch12/atm/ATMFrame.java (cont.)

```java
   /**
      Updates display message.
   */
   public void showState()
   {
      int state = theATM.getState();
      pad.clear();
      if (state == ATM.START)
         display.setText("Enter customer number\nA = OK");
      else if (state == ATM.PIN)
         display.setText("Enter PIN\nA = OK");
      else if (state == ATM.ACCOUNT)
         display.setText("Select Account\n"
               + "A = Checking\nB = Savings\nC = Exit");
      else if (state == ATM.TRANSACT)
         display.setText("Balance = "
               + theATM.getBalance()
               + "\nEnter amount and select transaction\n"
               + "A = Withdraw\nB = Deposit\nC = Cancel");
   }

```

Continued

ch12/atm/ATMFrame.java (cont.)

```
   class AButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         int state = theATM.getState();
         if (state == ATM.START)
            theATM.setCustomerNumber((int) pad.getValue());
         else if (state == ATM.PIN)
            theATM.selectCustomer((int) pad.getValue());
         else if (state == ATM.ACCOUNT)
            theATM.selectAccount(ATM.CHECKING);
         else if (state == ATM.TRANSACT)
         {
            theATM.withdraw(pad.getValue());
            theATM.back();
         }
         showState();
      }
   }

```

Continued

ch12/atm/ATMFrame.java (cont.)

```java
   class BButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         int state = theATM.getState();
         if (state == ATM.ACCOUNT)
            theATM.selectAccount(ATM.SAVINGS);
         else if (state == ATM.TRANSACT)
         {
            theATM.deposit(pad.getValue());
            theATM.back();
         }
         showState();
      }
   }

```

Continued

ch12/atm/ATMFrame.java (cont.)

```java
   class CButtonListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         int state = theATM.getState();
         if (state == ATM.ACCOUNT)
            theATM.reset();
         else if (state == ATM.TRANSACT)
            theATM.back();
         showState();
      }
   }
}
```

ch12/atm/KeyPad.java

```java
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
   A component that lets the user enter a number, using
   a button pad labeled with digits.
*/
public class KeyPad extends JPanel
{
   private JPanel buttonPanel;
   private JButton clearButton;
   private JTextField display;

```

Continued

ch12/atm/KeyPad.java (cont.)

```
      // Add digit buttons

      addButton("7");
      addButton("8");
      addButton("9");
      addButton("4");
      addButton("5");
      addButton("6");
      addButton("1");
      addButton("2");
      addButton("3");
      addButton("0");
      addButton(".");

      // Add clear entry button

      clearButton = new JButton("CE");
      buttonPanel.add(clearButton);

```

Continued

ch12/atm/KeyPad.java (cont.)

```
      class ClearButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            display.setText("");
         }
      }
      ActionListener listener = new ClearButtonListener();

      clearButton.addActionListener(new
            ClearButtonListener());

      add(buttonPanel, "Center");
   }

```

Continued

ch12/atm/KeyPad.java (cont.)

```
   /**
      Adds a button to the button panel
      @param label the button label
   */
   private void addButton(final String label)
   {
      class DigitButtonListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {

            // Don&apos;t add two decimal points
            if (label.equals(".")
                  && display.getText().indexOf(".") != -1)
               return;

            // Append label text to button
            display.setText(display.getText() + label);
         }
      }

```

Continued

ch12/atm/KeyPad.java (cont.)

```
      JButton button = new JButton(label);
      buttonPanel.add(button);
      ActionListener listener = new DigitButtonListener();
      button.addActionListener(listener);
   }

   /**
      Gets the value that the user entered.
      @return the value in the text field of the keypad
   */
   public double getValue()
   {
      return Double.parseDouble(display.getText());
   }

   /**
      Clears the display.
   */
   public void clear()
   {
      display.setText("");
   }
}
```

Self Check 12.12

Why does the `Bank` class in this example not store an array list of bank accounts?

Self Check 12.13

Suppose the requirements change — you need to save the current account balances to a file after every transaction and reload them when the program starts. What is the impact of this change on the design?

Big Java by Cay Horstmann