Chapter 7
Object-Oriented Design

Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus
Object-Oriented Design

• Now we can extend our discussion of the design of classes and objects

• Chapter 7 focuses on:
  – software development activities
  – determining the classes and objects that are needed for a program
  – the relationships that can exist among classes
  – the static modifier
  – writing interfaces
  – the design of enumerated type classes
  – method design and method overloading
  – GUI design and layout managers
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Program Development

• The creation of software involves four basic activities:
  – establishing the requirements
  – creating a design
  – implementing the code
  – testing the implementation

• These activities are not strictly linear – they overlap and interact
Requirements

• Software requirements specify the tasks that a program must accomplish
  – what to do, not how to do it

• Often an initial set of requirements is provided, but they should be critiqued and expanded

• It is difficult to establish detailed, unambiguous, and complete requirements

• Careful attention to the requirements can save significant time and expense in the overall project
MOBAS

Ospedale-Amico Freundliche-Klinik

Please enter your user code.
Bitte geben Sie Ihren Benutzer-Code
Prego inserire il proprio codice.

Login

v3.1

Messaggio

Pablo, è ora della sua prossima attività (Esame del sangue). Per cortesia, vada nella stanza 2.622.

OK

Hodgkin

Nuovi contenuti: Informazioni introduttive

1. COS’È

Il linfoma è un tumore che prende origine dalle ghiandole linfatiche, ovvero dalle cellule contenute nei tessuti (presenti in tutto il corpo) che hanno la funzione di difendere l’organismo dagli agenti esterni e dalle malattie.

Non è una singola malattia ma un gruppo eterogeneo con caratteristiche e storia clinica differenti. In generale si suddividono in Linfomi di Hodgkin (LH) e Linfomi non Hodgkin.

Sono chiari i contenuti nuovi che hai appena letto?

No    Un po’    Si

Continue
Gaius Julius Caesar

Divide et Impera
Design

- A *software design* specifies **how** a program will accomplish its requirements

- A software design specifies how the solution can be *broken down into manageable pieces* and what each piece will do

- An object-oriented design determines which *classes* and *objects* are needed, and specifies how they will interact

- **Low level** design details include how *individual methods* will accomplish their tasks
MOBAS Architecture

Client (Smartphone, Tablet, PC, Wall)

- HTML5/CSS3
- jQueryMobile (Javascript Framework)
- Browser (Firefox, Safari, Chrome, ...)
- OS (iOS, Android, ...)

Server

- LIFERAY PORTAL Portlet Container
- MOBAS Portlets
- External Components
  - Messages
  - MAP
  - PHIS
  - Charts
  - Mobas JQM
- TOMCAT Server
- JSP/Servlet
- JDBC
- JVM
- DB (PostgreSQL)
- OS
Implementation

- *Implementation* is the process of translating a design into source code

- *Novice programmers* often think that writing code is the heart of software development, but actually it should be the least creative step

- Almost all important decisions are made during requirements and design stages

- Implementation should focus on coding details, including style guidelines and documentation
Testing

• Testing attempts to ensure that the program will solve the intended problem under all the constraints specified in the requirements

• A program should be thoroughly tested with the goal of finding errors

• Debugging is the process of determining the cause of a problem and fixing it

• We revisit the details of the testing process later in this chapter
Quiz

During program development, software requirements specify:

A) how the program will accomplish the task
B) what the task is that the program must perform
C) how to divide the task into subtasks
D) how to test the program when it is done
E) all of the above
Quiz

During program development, software requirements specify:
A) how the program will accomplish the task
B) what the task is that the program must perform
C) how to divide the task into subtasks
D) how to test the program when it is done
E) all of the above

The specification phase is to understand the problem at hand so that the programmer can determine what needs to be done to solve the problem. The other efforts listed above are part of the design phase (A, C) and testing phase (D).
Quiz

Of the various phases in software development, which of the following is usually the lengthiest?

A) specification
B) design
C) implementation
D) testing
E) maintenance
Quiz

Of the various phases in software development, which of the following is usually the lengthiest?

A) specification
B) design
C) implementation
D) testing
E) maintenance

The maintenance phase exists for as long as the software is in use. Software requires modification (such as new requirements such as new features or I/O specifications) and so the maintenance phase is on-going whereas the other phases end once the software has been released and is in use.
A bad programming habit is to build an initial program and then spend a great deal of time modifying the code until it is acceptable. This is known as

A) the prototyping approach
B) the waterfall model
C) iterative development
D) the recursive approach
E) the build-and-fix approach
A bad programming habit is to build an initial program and then spend a great deal of time modifying the code until it is acceptable. This is known as
A) the prototyping approach
B) the waterfall model
C) iterative development
D) the recursive approach
E) the build-and-fix approach

Programmers who do not think things through will often build a program that does not fit the original requirements. They then spend an inordinate amount of time trying to repair the program to more properly fit. This is known as "build-and-fix" and is a poor programming practice.
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Identifying Classes and Objects

• The core activity of object-oriented design is determining the classes and objects that will make up the solution

• The classes may be part of a class library, reused from a previous project, or newly written

• One way to identify potential classes is to identify the objects discussed in the requirements

• Objects are generally nouns, and the services that an object provides are generally verbs
Identifying Classes and Objects

• A partial requirements document:

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

• Of course, not all nouns will correspond to a class or object in the final solution.
Identifying Classes and Objects

• Remember that a class represents a **group** (classification) of objects with the **same behaviors**

• Generally, classes that represent objects should be given names that are singular nouns

• **Examples:** Coin, Student, Message

• A class represents the concept of one such object

• We are free to instantiate as many of each object as needed
Identifying Classes and Objects

• Sometimes it is challenging to decide whether something should be represented as a class.

• For example, should an employee's address be represented as a set of *instance variables* or as an *Address object*?

• The more you examine the problem and its details the more clear these issues become.

• *When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities*
Identifying Classes and Objects

• We want to define classes with the proper amount of detail

• For example, it may be unnecessary to create separate classes for each type of appliance in a house

• It may be sufficient to define a more general Appliance class with appropriate instance data

• It all depends on the details of the problem being solved
Identifying Classes and Objects

• Part of identifying the classes we need is the process of assigning responsibilities to each class

• Every activity that a program must accomplish must be represented by one or more methods in one or more classes

• We generally use verbs for the names of methods

• In early stages it is not necessary to determine every method of every class – begin with primary responsibilities and evolve the design
Quiz

The idea that an object can exist separate from the executing program that creates it is called
A) transience
B) static
C) persistence
D) serialization
E) finality
Quiz

The idea that an object can exist separate from the executing program that creates it is called
A) transience
B) static
C) persistence
D) serialization
E) finality

Objects are stored in memory and are reclaimed by the garbage collector when they are no longer referenced or when a Java program terminates. It is desirable, however, to be able to save any given object for future use. This trait is called persistence, and the ability to do this is by saving the instance data of the object to a file.
Quiz

In which phase of program development would you expect the programmer(s) to create the pseudocode (e.g. a flowchart of the main algorithm)?

A) Software requirements
B) Software design
C) Software implementation
D) Software testing
E) Could occur in any of the above
Quiz

In which phase of program development would you expect the programmer(s) to create the pseudocode?
A) Software requirements
B) Software design
C) Software implementation
D) Software testing
E) Could occur in any of the above

Pseudocode is a description of an algorithm written in an English-like way rather than in a specific programming language. This is part of the program's design. In the implementation phase, the programmer(s) translates the pseudocode into the programming language being used.
Quiz

• Analyze the following requirements and identify possible classes

• The user must be allowed to select one category among: events, hotels, attractions, suggestions. Searching for hotels the user may specify the booking period, the required hotel facilities and preferred cost. Results could be ordered by distance to a target position (e.g., the current one), by price, by the system predicted user utility. Attractions are searched by tags: the system provides the attractions that have been tagged with a user specified tag.
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Static Class Members

• Recall that a **static method** is one that can be invoked through its class name

• For example, the methods of the **Math class** are static:

  \[
  \text{result} = \text{Math.sqrt}(25)
  \]

• **Variables** can be **static** as well

• **Determining** if a method or variable should be static is an important design decision
The static Modifier

• We declare **static methods and variables** using the **static** modifier

• **It associates** the method or variable **with the class** rather than with an object of that class

• Static methods are sometimes called **class methods** and static variables are sometimes called **class variables**

• Let's carefully consider the implications of each
Static variable example

```java
class Pendulum {  
    float mass;  
    float length;  
    int cycles;  
    position ();  
    static float gravAccel = 9.8;  
}

class TextBook {  
    Pendulum bigPendulum;  
    Pendulum smallPendulum;  
    Pendulum bigPendulum = new Pendulum();  
    Pendulum smallPendulum = new Pendulum();  
    Pendulum smallPendulum = new Pendulum();  
    static float gravAccel = 9.8;  
}
```
Static Variables

• Normally, each object has its own data space, but if a variable is declared as static, **only one copy of the variable exists**

```java
private static float price;
```

• **Memory space for a static variable** is created when the **class is first referenced**

• **All objects** instantiated from the class **share its static variables**

• **Changing** the value of a static variable **in one object changes it for all others**
Quiz

• Assume that you are defining a `BankAccount` class whose objects each represent a separate bank account. Write a `declaration` for a `variable` that can hold the combined total balance of all the bank accounts objects.
Quiz

• Assume that you are defining a BankAccount class whose objects each represent a separate bank account. Write a declaration for a variable that can hold the combined total balance of all the bank accounts objects.

• private static double totalBalance = 0;
Static Methods

```java
public class Helper {
    public static int cube (int num) {
        return num * num * num;
    }
}
```

- Because it is declared as static, the `cube` method can be invoked through the class name:

  ```java
  value = Helper.cube(4);
  ```
Quiz

Which of the following methods is a static method? The class in which the method is defined is given in parentheses following the method name.

A) equals (String)
B) toUpperCase (String)
C) showInputDialog (JOptionPane)
D) format (DecimalFormat)
E) paint (Applet)
Quiz

Which of the following methods is a static method? The class in which the method is defined is given in parentheses following the method name.

A) equals (String)
B) toUpperCase (String)
C) showInputDialog (JOptionPane)
D) format (DecimalFormat)
E) paint (Applet)

The JOptionPane class defines several of its methods to be static. Invoking these methods is done by using JOptionPane rather than a variable of type JOptionPane. The other methods above are not static.
Static Class Members (data and methods)

• The order of the modifiers can be interchanged, but by convention visibility modifiers come first.

• Recall that the main method is static – it is invoked by the Java interpreter without creating an object.

• Static methods cannot reference instance variables because instance variables don't exist until an object exists.

• However, a static method can reference static variables or local variables (or static methods).
Quiz

• Can you find any syntax error here?

```java
public class StaticEx {
    private int bad;
    public StaticEx(int inputBad) {
        bad = inputBad;
    }
    public static void main(String[] args) {
        int bad = (int) Math.random() * 6;
        System.out.println("Bad is: " + bad);
        StaticEx objBad = new StaticEx(3);
        System.out.println("Bad in object is: " + objBad.bad);
    }
}
```
Quiz

• Can you find any syntax error here?

```java
public class StaticEx {
    private int bad;
    public StaticEx(int input) {
        bad = input;
    }
    public static void main(String[] args) {
        bad = (int) Math.random() * 6;
        System.out.println("Bad is: " + bad);
        StaticEx objBad = new StaticEx(3);
        System.out.println("Bad in object is: " + objBad.bad);
    }
}
```

In a static method of a class (e.g., main) you cannot use an instance variable (non static), unless you have a reference to an object of that class.

This is legal
Static Class Members

• Static methods and static variables often work together

• The following example keeps track of how many Slogan objects have been created using a static variable, and makes that information available using a static method

• See SloganCounter.java
• See Slogan.java
public class SloganCounter {

    //---
    //  Creates several Slogan objects and prints the number of
    //  objects that were created.
    //---
    public static void main (String[] args) {
        Slogan obj;

        obj = new Slogan("Remember the Alamo.");
        System.out.println(obj);

        obj = new Slogan("Don't Worry. Be Happy.");
        System.out.println(obj);
    }

    continue
continue

    obj = new Slogan ("Live Free or Die.");
    System.out.println (obj);

    obj = new Slogan ("Talk is Cheap.");
    System.out.println (obj);

    obj = new Slogan ("Write Once, Run Anywhere.");
    System.out.println (obj);

    System.out.println();
    System.out.println ("Slogans created: " + Slogan.getCount());
}
```java
obj = new Slogan("Live Free or Die.");
System.out.println(obj);
obj = new Slogan("Talk is Cheap.");
System.out.println(obj);
obj = new Slogan("Write Once, Run Anywhere.");
System.out.println(obj);
System.out.println();
System.out.println("Slogans created: " + Slogan.getCount());
```
Quiz

• Without looking at the next slide – write down the core code of the Slogan class (instance and class variables and constructor)

• Hints
  – one (instance or class ?) variable (count) for storing the number of slogans created and one for the slogan string
  – the code required for updating the count variable must be in the constructor of the Slogan class
public class Slogan
{
    private String phrase;
    private static int count = 0;

    // Constructor: Sets up the slogan and counts the number of instances created.
    public Slogan (String str)
    {
        phrase = str;
        count++;
    }
}

continue
continue

// Returns this slogan as a string.
public String toString()
{
    return phrase;
}

// Returns the number of instances of this class that have been created.
public static int getCount ()
{
    return count;
}
}
Quick Check

Why can't a static method reference an instance variable?
Quick Check

Why can't a static method reference an instance variable?

Because instance data is created only when an object is created.

You don't need an object to execute a static method.

And even if you had an object, which object's instance data would be referenced? (remember, the method is invoked through the class name)
Quiz

• What kinds of variables and methods can the main method of any class reference? Why?
Quiz

• What kinds of variables and methods can the main method of any class reference? Why?

• The main method is static, and therefore can only refer to static or local variables and static methods. The main method cannot refer to instance variables and non static methods (unless an object of the class is created and the method or the instance data is accessed through a reference to the object).
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Class Relationships

• Classes in a software system can have various types of relationships to each other

• Three of the most common relationships:
  – **Dependency**: A uses B
  – **Aggregation**: A has-a B
  – **Inheritance**: A is-a B

• Let's discuss dependency and aggregation further

• Inheritance is discussed in detail in Chapter 9
Dependency

• A dependency exists when one class relies on another in some way, usually by invoking the methods of the other.

• We've seen dependencies in many previous examples.

• We don't want numerous or complex dependencies among classes.

• Nor do we want complex classes that don't depend on others.

• A good design strikes the right balance.
Dependency

• Some dependencies occur between objects of the same class

• A method of the class may accept an object of the same class as a parameter

• For example, the `concat` method of the `String` class takes as a parameter another `String` object

\[ \text{str3} = \text{str1.concat(str2)}; \]
Dependency

• The following example defines a class called RationalNumber

• A rational number is a value that can be represented as the ratio of two integers

• Several methods of the RationalNumber class accept another RationalNumber object as a parameter

• See RationalTester.java
• See RationalNumber.java
public class RationalTester {
    public static void main (String[] args) {
        RationalNumber r1 = new RationalNumber (6, 8);
        RationalNumber r2 = new RationalNumber (1, 3);
        RationalNumber r3, r4, r5, r6, r7;

        System.out.println ("First rational number: " + r1);
        System.out.println ("Second rational number: " + r2);
        continue
if (r1.isLike(r2))
    System.out.println("r1 and r2 are equal.");
else
    System.out.println("r1 and r2 are NOT equal.");

r3 = r1.reciprocal();
System.out.println("The reciprocal of r1 is: " + r3);

r4 = r1.add(r2);
r5 = r1.subtract(r2);
r6 = r1.multiply(r2);
r7 = r1.divide(r2);

System.out.println("r1 + r2: " + r4);
System.out.println("r1 - r2: " + r5);
System.out.println("r1 * r2: " + r6);
System.out.println("r1 / r2: " + r7);
if (r1.isLike(r2))
    System.out.println("r1 and r2 are equal.");
else
    System.out.println("r1 and r2 are NOT equal.");

r3 = r1.reciprocal();
System.out.println("The reciprocal of r1 is: " + r3);

r4 = r1.add(r2);
r5 = r1.subtract(r2);
r6 = r1.multiply(r2);
r7 = r1.divide(r2);

System.out.println("r1 + r2: " + r4);
System.out.println("r1 - r2: " + r5);
System.out.println("r1 * r2: " + r6);
System.out.println("r1 / r2: " + r7);
public class RationalNumber
{
    private int numerator, denominator;

    public RationalNumber (int numer, int denom)
    {
        if (denom == 0)
            denom = 1;

        // Make the numerator "store" the sign
        if (denom < 0)
        {
            numer = numer * -1;
            denom = denom * -1;
        }
    }
continue

numerator = numer;
denominator = denom;

reduce();
}

//-----------------------------------------------------------------
//  Returns the numerator of this rational number.
//-----------------------------------------------------------------
public int getNumerator ()
{
    return numerator;
}

//-----------------------------------------------------------------
//  Returns the denominator of this rational number.
//-----------------------------------------------------------------
public int getDenominator ()
{
    return denominator;
}
// Returns the reciprocal of this rational number.
public RationalNumber reciprocal ()
{
    return new RationalNumber (denominator, numerator);
}

// Adds this rational number to the one passed as a parameter.
// A common denominator is found by multiplying the individual denominators.
public RationalNumber add (RationalNumber op2)
{
    int commonDenominator = denominator * op2.getDenominator();
    int numerator1 = numerator * op2.getDenominator();
    int numerator2 = op2.getNumerator() * denominator;
    int sum = numerator1 + numerator2;

    return new RationalNumber (sum, commonDenominator);
}
public RationalNumber subtract (RationalNumber op2) {
    int commonDenominator = denominator * op2.getDenominator();
    int numerator1 = numerator * op2.getDenominator();
    int numerator2 = op2.getNumerator() * denominator;
    int difference = numerator1 - numerator2;

    return new RationalNumber (difference, commonDenominator);
}

public RationalNumber multiply (RationalNumber op2) {
    int numer = numerator * op2.getNumerator();
    int denom = denominator * op2.getDenominator();

    return new RationalNumber (numer, denom);
}
```java
// Divides this rational number by the one passed as a parameter
// by multiplying by the reciprocal of the second rational.
public RationalNumber divide (RationalNumber op2) {
    return multiply (op2.reciprocal());
}

// Determines if this rational number is equal to the one passed
// as a parameter. Assumes they are both reduced.
public boolean isLike (RationalNumber op2) {
    return (numerator == op2.getNumerator() &&
            denominator == op2.getDenominator());
}
```
public String toString ()
{
    String result;
    if (numerator == 0)
        result = "0";
    else
        if (denominator == 1)
            result = numerator + ";
        else
            result = numerator + "/" + denominator;
    return result;
}
private void reduce ()
{
    if (numerator != 0)
    {
       int common = gcd (Math.abs(numerator), denominator);

       numerator = numerator / common;
       denominator = denominator / common;
    }
}

continue
// Computes and returns the greatest common divisor of the two positive parameters. Uses Euclid's algorithm.
private int gcd (int num1, int num2) {
    while (num1 != num2) {
        if (num1 > num2)
            num1 = num1 - num2;
        else
            num2 = num2 - num1;
    }
    return num1;
}
Exercise

• Draw the flowchart of the gcd method shown in the previous slide.
Start gcd(a, b)

a /= b

false

ture

true

a > b

false

ture

a = a - b

b = b - a

return a
Quiz

• **Rewrite the `toString()` method of the class `RationalNumber` using the conditional operator**

```java
public String toString() {
    String result;
    if (numerator == 0) {
        result = "0";
    } else if (denominator == 1) {
        result = numerator + "";
    } else {
        result = numerator + "/" + denominator;
    }
    return result;
}
```
Quiz

• Rewrite the `toString()` method of the class `RationalNumber` using the conditional operator

```java
public String toString() {
    return (numerator == 0) ? "0" : (denominator == 1)
        ? numerator + "" : numerator + "/"
        + denominator;
}
```
Quiz

The method `reciprocal` returns a new `RationalNumber` object. Write the code of a method of the class `RationalNumber` that transforms a non-zero `RationalNumber` object into its reciprocal (it changes the state of the object).
Quiz

The method reciprocal returns a new RationalNumber object. Write the code of a new method of the class RationalNumber that transforms a RationalNumber object into its reciprocal (it changes the state of the object).

```java
public void setReciprocal() {
    if (numerator != 0) {
        int temp = numerator;
        numerator = denominator;
        denominator = temp;
        if (denominator < 0) {
            numerator = numerator * -1;
            denominator = denominator * -1;
        }
    }
}
```
Aggregation

- An **aggregate** is an object that is *made up* of other objects.
- An aggregate object contains references to other objects as **instance data**.
- Therefore aggregation is a *has-a* relationship.
  - A family *has* some members.
- This is a **special kind of dependency**; the aggregate relies on the objects that compose it.
Aggregation

• In the following example, a Student object is composed, in part, of Address objects

• A student has an address (in fact each student has two addresses)

• The Student class aggregates Address(es)

• See StudentBody.java
• See Student.java
• See Address.java
public class StudentBody
{
    //---
    //  Creates some Address and Student objects and prints them.
    //---
    public static void main (String[] args)
    {
        Address school = new Address ("800 Lancaster Ave.", "Villanova", "PA", 19085);
        Address jHome = new Address ("21 Jump Street", "Lynchburg", "VA", 24551);
        Student john = new Student ("John", "Smith", jHome, school);
        Address mHome = new Address ("123 Main Street", "Euclid", "OH", 44132);
        Student marsha = new Student ("Marsha", "Jones", mHome, school);

        System.out.println (john);
        System.out.println (marsha);
    }
}
public class StudentBody {
    // Creates some Address and Student objects and prints them.
    public static void main (String[] args) {
        Address school = new Address("800 Lancaster Ave.", "Villanova", "PA", 19085);
        Address jHome = new Address("21 Jump Street", "Lynchburg", "VA", 24551);
        Student john = new Student("John", "Smith", jHome, school);
        Address mHome = new Address("123 Main Street", "Euclid", "OH", 44132);
        Student marsha = new Student("Marsha", "Jones", mHome, school);
        System.out.println (john);
        System.out.println ();
        System.out.println (marsha);
    }
}
public class Student
{
    private String firstName, lastName;
    private Address homeAddress, schoolAddress;

    // Constructor: Sets up this student with the specified values.
    public Student (String first, String last, Address home, Address school)
    {
        firstName = first;
        lastName = last;
        homeAddress = home;
        schoolAddress = school;
    }
}

continue
// Returns a string description of this Student object.
public String toString()
{
    String result;
    result = firstName + " " + lastName + "\n";
    result += "Home Address: \n" + homeAddress + "\n";
    result += "School Address: \n" + schoolAddress;
    return result;
}
public class Address
{
    private String streetAddress, city, state;
    private long zipCode;

    // Constructor: Sets up this address with the specified data.
    public Address (String street, String town, String st, long zip)
    {
        streetAddress = street;
        city = town;
        state = st;
        zipCode = zip;
    }
}

continue
public String toString()
{
    String result;

    result = streetAddress + "\n";
    result += city + ", " + state + " " + zipCode;

    return result;
}
Aggregation in UML

StudentBody

+ main (args : String[]) : void

Student

- firstName : String
- lastName : String
- homeAddress : Address
- schoolAddress : Address

+ toString() : String

Address

- streetAddress : String
- city : String
- state : String
- zipCode : long

+ toString() : String

Copyright © 2012 Pearson Education, Inc.
The this Reference

• The this reference allows an object to refer to itself
• That is, the this reference, used inside a method, refers to the object through which the method is being executed
• Suppose the this reference is used inside a method called tryMe, which is invoked as follows:

```java
obj1.tryMe();
obj2.tryMe();
```

• In the first invocation, the this reference refers to obj1; in the second it refers to obj2
The this reference

• The this reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names.

• The constructor of the Account class from Chapter 4 could have been written as follows:

```java
public Account (String name, long acctNumber, double balance) {
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
}
```
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Interfaces

- A Java interface is a collection of abstract methods and constants

- An abstract method is a method header without a method body (!?)

- An abstract method can be declared using the modifier abstract, but because all methods in an interface are abstract, usually it is left off

- An interface is used to establish a set of methods that a class will implement
public interface Doable
{
    public void doThis();
    public int doThat();
    public void doThis2 (double value, char ch);
    public boolean doTheOther (int num);
}

A semicolon immediately follows each method header

interface is a reserved word

None of the methods in an interface are given a definition (body)
Interfaces

- An interface **cannot be instantiated**

- **Methods** in an interface have **public** visibility by default (*they cannot be private*)

- A class formally implements an interface by:
  - stating so in the class header
  - providing **implementations** for every abstract method in the interface

- If a class declares that it implements an interface, it **must** define all methods in the interface
**Interfaces**

```java
class CanDo implements Doable {
    public void doThis () {
        // whatever
    }
    public void doThat () {
        // whatever
    }
    // etc.
}
```

*implments is a reserved word*

- Each method listed in `Doable` is given a definition
Quiz

• Write the code of the interface ActionListener (in accordance to how you have it used it so far)
Quiz

• Write the code of the interface ActionListener (in accordance to how you have it used it so far)

    public interface ActionListener {

        public void actionPerformed (ActionEvent event);

    }
Interfaces

• In addition to (or instead of) abstract methods, an interface can contain constants (public static final)

• For instance `Transparency`

```java
package java.awt;

public interface Transparency {
    public static final int OPAQUE = 1;
    public static final int BITMASK = 2;
    public static final int TRANSLUCENT = 3;
    public int getTransparency();
}
```
Interfaces

• When a class implements an interface, it gains access to all its constants:

```java
public class Layer implements Transparency {}
```

Then you can evaluate: `Layer.OPAQUE`

• A class that implements an interface can implement other methods as well

• See `Complexity.java`
• See `Question.java`
• See `MiniQuiz.java`
UML diagram

```
<<Interface>>
Complexity

+ getComplexity() : int
+ setComplexity(int) : void

MiniQuiz

+ main(args : String[]) : void

Question

+ getQuestion() : String
+ getAnswer() : String
+ answerCorrect(String) : boolean
+ toString() : String
```
public interface Complexity
{
    public void setComplexity (int complexity);
    public int getComplexity();
}
public class Question implements Complexity {
    private String question, answer;
    private int complexityLevel;

    // Constructor: Sets up the question with a default complexity.
    public Question (String query, String result) {
        question = query;
        answer = result;
        complexityLevel = 1;
    }
}
continue
public void setComplexity (int level)
{
    complexityLevel = level;
}

public int getComplexity()
{
    return complexityLevel;
}

public String getQuestion()
{
    return question;
}
public String getAnswer()
{
    return answer;
}

public boolean answerCorrect(String candidateAnswer)
{
    return answer.equals(candidateAnswer);
}

public String toString()
{
    return question + "\n" + answer;
}
import java.util.Scanner;

public class MiniQuiz
{
    public static void main (String[] args)
    {
        Question q1, q2;
        String possible;

        Scanner scan = new Scanner (System.in);

        q1 = new Question ("What is the capital of Jamaica?", "Kingston");
        q1.setComplexity (4);

        q2 = new Question ("Which is worse, ignorance or apathy?", "I don't know and I don't care");
        q2.setComplexity (10);

        continue
continue

    System.out.print (q1.getQuestion());
    System.out.println(" (Level: " + q1.getComplexity() + ")");
    possible = scan.nextLine();
    if (q1.answerCorrect(possible))
        System.out.println("Correct");
    else
        System.out.println("No, the answer is " + q1.getAnswer());

    System.out.println();
    System.out.print (q2.getQuestion());
    System.out.println(" (Level: " + q2.getComplexity() + ")");
    possible = scan.nextLine();
    if (q2.answerCorrect(possible))
        System.out.println("Correct");
    else
        System.out.println("No, the answer is " + q2.getAnswer());
Sample Run

What is the capital of Jamaica? (Level: 4)
Kingston
Correct

Which is worse, ignorance or apathy? (Level: 10)
apathy
No, the answer is I don't know and I don't care
Interfaces

- **Multiple classes** can implement the **same** interface
- **A class** can implement **multiple interfaces**
- The interfaces are listed in the `implements` clause
- The class **must** implement all methods in all interfaces listed in the header

```java
class ManyThings implements interface1, interface2 {
    // all methods of both interfaces
}
```
Interfaces

• The Java API contains many helpful interfaces

• The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects

• We discussed the `compareTo` method of the `String` class in Chapter 5

• The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order
The Comparable Interface

• Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```java
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

• The value returned from `compareTo` should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`

• It's up to the programmer to determine what makes one object less than another
Quiz

Implement the `compareTo` method for the class `RationalNumber`

```java
public class RationalNumber {
    private int numerator, denominator;
    ...
}
```
Quiz

Implement the `compareTo` method for the class `RationalNumber`

```java
public class RationalNumber {
    private int numerator, denominator;
    ...
}

public int compareTo (RationalNumber num2) {
    RationalNumber diff = subtract(num2);
    return diff.getNumerator();
}
```
The Iterator Interface

• As we discussed in Chapter 5, an iterator is an object that provides a means of processing a collection of objects one at a time

• An iterator is created formally by implementing the `Iterator` interface, which contains three methods

  – The `hasNext` method returns a boolean result – true if there are items left to process

  – The `next` method returns the next object in the iteration

  – The `remove` method removes the object most recently returned by the `next` method (*it is optional*)
The Iterable Interface

• Another interface, `Iterable`, establishes that an object provides an iterator

• The `Iterable` interface (e.g. `ArrayList`) has one method, `iterator()`, that returns an `Iterator` object

• Any `Iterable` object can be processed using the for-each version of the `for` loop

• Note the difference: an `Iterator` has methods that perform an iteration; an `Iterable` object provides an iterator on request
Interfaces

• You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (Comparable)

• However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways

• Interfaces are a key aspect of object-oriented design in Java

• We discuss this idea further in Chapter 10 (Polymorphism)
Quiz

• Define a Java interface called Nameable. Classes implementing this interface must provide a setName method that requires a String parameter and returns nothing, and a getName method that has no parameters and returns a String.
Quiz

• Define a Java interface called Nameable. Classes implementing this interface must provide a setName method that requires a String parameter and returns nothing, and a getName method that has no parameters and returns a String.

```java
public interface Nameable {
    public void setName(String name);
    public String getName();
}
```
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Enumerated Types

• In Chapter 3 we introduced enumerated types, which define a new data type and list all possible values of that type:

```c
enum Season {winter, spring, summer, fall}
```

• Once established, the new type can be used to declare variables

```c
Season time;
```

• The only values this variable can be assigned are the ones established in the `enum` definition
Enumerated Types

• An enumerated type definition is a special kind of class

• The values of the enumerated type are objects of that type

• For example, fall is an object of type Season

• That's why the following assignment is valid:

      time = Season.fall;
Enumerated Types

• An enumerated type definition can be more interesting than a simple list of values.

• Because they are like classes, we can add additional instance data and methods.

• We can define an enum constructor as well.

• Each value listed for the enumerated type calls the constructor.

• See Season.java

• See SeasonTester.java
public enum Season {
    winter ("December through February"),
    spring ("March through May"),
    summer ("June through August"),
    fall ("September through November");

    private String span;

    continue

These are the objects of the Season class

This is the only way to create objects of type Season because an enum cannot be instantiated.
Season (String months)
{
    span = months;
}

public String getSpan()
{
    return span;
}
public class SeasonTester {
    public static void main (String[] args) {
        for (Season time : Season.values())
            System.out.println (time + "\t" + time.getSpan());
    }
}
public class SeasonTester {
    
    // Iterates through the values of the Season enumerated type.
    public static void main (String[] args) {
        for (Season time : Season.values())
            System.out.println (time + "\t" + time.getSpan());
    }
}
Enumerated Types

• Every enumerated type contains a static method called `values()` that returns a list of all possible values for that type

• The list returned from `values` can be processed using a for-each loop

• *An enumerated type cannot be instantiated outside of its own definition*

• A carefully designed enumerated type provides a versatile and type-safe mechanism for managing data
Exercise

• Write an enum that contains the 5 unibz faculties. Each faculty is described by name (String), rank (int) and numStudents (int).

• Write also a main method that print the faculty with the best rank and the faculty with the largest number of students.

• Hint: scan the faculties with a for each loop and store the faculty with rank = 1, and with the largest number of students, then print these to faculties (variables).
public enum Faculty {

    ComputerScience("Faculty of Computer Science", 1, 100),
    ScienceTechnology("Faculty of Science and Tecnology", 5, 50),
    Design("Faculty of Design", 2, 200),
    Education("Faculty of Education", 3, 500),
    Economics("Faculty of Economics", 2, 1000);

    String name;
    int rank;
    int numStudents;

    // enum constructors are private
    private Faculty(String name, int rank, int numStud) {
        this.name = name;
        this.rank = rank;
        this.numStudents = numStud;
    }

    public String toString() {
        return name;
    }

    ...
public static void main(String[] args) {

    Faculty bestRank = null;
    Faculty bestNum = null;
    int maxNumStud = 0;

    for (Faculty f : Faculty.values()) {
        if (f.rank == 1)
            bestRank = f;
        if (f.numStudents > maxNumStud)
            bestNum = f;
    }

    System.out.println("Best ranking faculty : " + bestRank);
    System.out.println("Largest faculty : " + bestNum);
}
}
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Method Design

• As we've discussed, high-level design issues include:
  – identifying primary classes and objects
  – assigning primary responsibilities

• After establishing high-level design issues, it is important to address low-level issues such as the design of key methods

• For some methods, careful planning is needed to make sure they contribute to an efficient and elegant system design
Method Decomposition

• A method should be relatively small, so that it can be understood as a single entity

• A potentially large method should be decomposed into several smaller methods as needed for clarity

• A public service method of an object may call one or more private support methods to help it accomplish its goal

• Support methods might call other support methods if appropriate
Method Decomposition

• Let's look at an example that requires method decomposition – translating English into Pig Latin

• Pig Latin is a language in which each word is modified by moving the initial sound of the word to the end and adding "ay"

• Words that begin with vowels have the "yay" sound added on the end

  book ➔ ookbay  table ➔ abletay

  item ➔ itemyay  chair ➔ airchay
Method Decomposition

• The primary objective (translating a sentence) is too complicated for one method to accomplish.

• Therefore we look for natural ways to decompose the solution into pieces.

• Translating a sentence can be decomposed into the process of translating each word.

• The process of translating a word can be separated into translating words that:
  – begin with vowels
  – begin with consonant blends (sh, cr, th, etc.)
  – begin with single consonants
Method Decomposition

• In a UML class diagram, the visibility of a variable or method can be shown using special characters

• Public members are preceded by a plus sign

• Private members are preceded by a minus sign

• **See** PigLatin.java
• **See** PigLatinTranslator.java
Class Diagram for Pig Latin

PigLatin

+ main (args : String[]) : void

PigLatinTranslator

+ translate (sentence : String) : String
- translateWord (word : String) : String
- beginsWithVowel (word : String) : boolean
- beginsWithBlend (word : String) : boolean
import java.util.Scanner;

public class PigLatin {
    //---
    // Reads sentences and translates them into Pig Latin.
    //---
    public static void main (String[] args) {
        String sentence, result, another;

        Scanner scan = new Scanner (System.in);

        continue
continue

do
{
    System.out.println ();
    System.out.println ("Enter a sentence (no punctuation): ");
    sentence = scan.nextLine();
    System.out.println ();
    result = PigLatinTranslator.translate (sentence);
    System.out.println ("That sentence in Pig Latin is: ");
    System.out.println (result);
    System.out.println ();
    System.out.println ();
    System.out.println ();
    System.out.println ("Translate another sentence (y/n)? ");
    another = scan.nextLine();
}
    while (another.equalsIgnoreCase("y"));
}
continue

do {
    System.out.println();
    System.out.println("Enter a sentence (no punctuation): ");
    sentence = scan.nextLine();
    System.out.println();
    result = PigLatinTranslator.translate(sentence);
    System.out.println("That sentence in Pig Latin is: ");
    System.out.println(result);
    System.out.println();
    System.out.print("Translate another sentence (y/n)? ");
    another = scan.nextLine();
} while (another.equalsIgnoreCase("y"));
}
import java.util.Scanner;

public class PigLatinTranslator
{

    // Translates a sentence of words into Pig Latin.
    public static String translate (String sentence)
    {
        String result = "";

        sentence = sentence.toLowerCase();

        Scanner scan = new Scanner (sentence);

        while (scan.hasNext())
        {
            result += translateWord (scan.next());
            result += " ";
        }

        return result;
    }
}

private String translateWord (String word)
{
    String result = "";

    if (word.length() > 1)
    {
        result += word.substring(1) + "a" + word.substring(0, 1);
    }
    else
    {
        result += word + "a";
    }

    return result;
}
private static String translateWord (String word) {
    String result = "";

    if (beginsWithVowel(word))
        result = word + "yay";
    else
        if (beginsWithBlend(word))
            result = word.substring(2) + word.substring(0,2) + "ay";
        else
            result = word.substring(1) + word.charAt(0) + "ay";

    return result;
}

// Translates one word into Pig Latin. If the word begins with a vowel, the suffix "yay" is appended to the word. Otherwise, the first letter or two are moved to the end of the word, and "ay" is appended.

continue
private static boolean beginsWithVowel (String word) {
    String vowels = "aeiou";

    char letter = word.charAt(0);

    return (vowels.indexOf(letter) != -1);  
}
continues

// Determines if the specified word begins with a particular
// two-character consonant blend.
//
private static boolean beginsWithBlend (String word)
{
    return word.startsWith("bl") || word.startsWith("sc") ||
        word.startsWith("br") || word.startsWith("sh") ||
        word.startsWith("ch") || word.startsWith("sk") ||
        word.startsWith("cl") || word.startsWith("sl") ||
        word.startsWith("cr") || word.startsWith("sn") ||
        word.startsWith("dr") || word.startsWith("sm") ||
        word.startsWith("dw") || word.startsWith("sp") ||
        word.startsWith("fl") || word.startsWith("sq") ||
        word.startsWith("fr") || word.startsWith("st") ||
        word.startsWith("gl") || word.startsWith("sw") ||
        word.startsWith("gr") || word.startsWith("th") ||
        word.startsWith("kl") || word.startsWith("tr") ||
        word.startsWith("ph") || word.startsWith("tw") ||
        word.startsWith("pl") || word.startsWith("wh") ||
        word.startsWith("pr") || word.startsWith("wr") ;
}
Quiz

• Consider the following two methods of String
  – public int `indexOf(String str)` : Returns the index within `this` string of the first occurrence of the specified substring (str), and -1 if str does not occur in this string.
  – public boolean `startsWith(String prefix)` : Tests if this string starts with the specified prefix.

• What is the relationship between these methods? Implement `startsWith` using `indexOf`
Quiz

• Consider the following two methods of String
  – public int indexOf(String str) : Returns the index within this string of the first occurrence of the specified substring (-1 if it does not occur).
  – public boolean startsWith(String prefix) : Tests if this string starts with the specified prefix.

• What is the relationship between these methods? Implement startsWith using indexOf

  public boolean startsWith(String start) {
      return ((this.indexOf(start) == 0) ? true : false);
  }

  or even shorter ...
Objects as Parameters

• Another important issue related to method design involves parameter passing

• Parameters in a Java method are **passed by value**

• A copy of the *actual parameter* (the value passed in) is stored into the *formal parameter* (in the method header)

• When an *object* is passed to a method, the *actual parameter* and the *formal parameter* become *aliases* of each other
Passing Objects to Methods

• What a method does with a parameter **may or may not have a permanent effect** (outside the method)

• Note the difference between changing the internal state of an object versus changing which object a reference points to

• See ParameterTester.java
• See ParameterModifier.java
• See Num.java
public class ParameterTester
{
    // Sets up three variables (one primitive and two objects) to
    // serve as actual parameters to the changeValues method. Prints
    // their values before and after calling the method.

    public static void main (String[] args)
    {
        ParameterModifier modifier = new ParameterModifier();

        int a1 = 111;
        Num a2 = new Num (222);
        Num a3 = new Num (333);

        continue
continue

System.out.println ("Before calling changeValues:");
System.out.println ("a1\ta2\ta3");
System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

modifier.changeValues (a1, a2, a3);

System.out.println ("After calling changeValues:");
System.out.println ("a1\ta2\ta3");
System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
}
Output

Before calling changeValues:
\n\n| a1 | a2 | a3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>222</td>
<td>333</td>
</tr>
</tbody>
</table>

Before changing the values:
\n\n| f1 | f2 | f3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>222</td>
<td>333</td>
</tr>
</tbody>
</table>

After changing the values:
\n\n| f1 | f2 | f3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>999</td>
<td>888</td>
<td>777</td>
</tr>
</tbody>
</table>

After calling changeValues:
\n\n| a1 | a2 | a3 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>888</td>
<td>333</td>
</tr>
</tbody>
</table>


/*****************************************************************************/
//  ParameterModifier.java       Author: Lewis/Loftus
//
//  Demonstrates the effects of changing parameter values.
/**************************************************************** ***********/

public class ParameterModifier
{

    //-------------------------------------------------------------------------------
    //  Modifies the parameters, printing their values before and
    //  after making the changes.
    //-------------------------------------------------------------------------------
    public void changeValues (int f1, Num f2, Num f3)
    {
        System.out.println ("Before changing the values:");
        System.out.println ("f1\t\tf2\t\tf3");
        System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

        f1 = 999;
        f2.setValue(888);
        f3 = new Num (777);

        System.out.println ("After changing the values:");
        System.out.println ("f1\t\tf2\t\tf3");
        System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
    }
}

//********************************************************************
// Num.java       Author: Lewis/Loftus
//
// Represents a single integer as an object.
//*********************************************************************

global class Num
{
    private int value;

    // Sets up the new Num object, storing an initial value.
    public Num (int update)
    {
        value = update;
    }

    continue
public void setValue (int update)
{
    value = update;
}

public String toString ()
{
    return value + "";
}
STEP 1
Before invoking changeValues

STEP 2
modifier. tester.changeValues (a1, a2, a3);

STEP 3
f1 = 999;

STEP 4
f2.setValue (888);

STEP 5
f3 = new Num (777);

STEP 6
After returning from changeValues
Quiz

Write a static method of the class Die:
public static int rollIt(Die die)
that rolls the Die passed as parameter (change the state) and returns the new faceValue
Quiz

Write a static method of the class Die

\[
\text{public static int rollIt(Die die)}
\]

that rolls the Die passed as parameter (change the state) and returns the new faceValue

\[
\text{public static int rollIt(Die d) \{} \\
\quad \text{return d.roll();} \\
\text{\}}
\]
Circle

- Define a class Circle with instance data xCenter, yCenter, and radius (all int)
- Write a constructor that initializes the instance data
- Write a method of the class Circle that changes the centre of a Circle object by adding a deltaX (parameter) and deltaY (parameter) to the coordinates of the center
public class Circle {

    private int xCenter, yCenter, radius;

    Circle(int xCenter, int yCenter, int radius) {
        this.xCenter = xCenter;
        this.yCenter = yCenter;
        this.radius = radius;
    }

    public void moveCenter(int xMove, int yMove) {
        xCenter += xMove;
        yCenter += yMove;
    }

}
Circle II

• Write a toString() method for the Circle class and a main method that create an object of this class and then move its centre.

• Use the method toString() to show the state of the object before and after the movement of the centre.
public String toString() {
    return "Circle: x = " + xCenter + "; y = " + yCenter + "; r = " + radius;
}

public static void main(String[] args) {
    Circle c = new Circle(1,1, 4);
    System.out.println(c);
    c.moveCenter(3, 2);
    System.out.println(c);
}
Circle III

• Write a static method of the class Circle that takes as parameters a Circle and two int (xMove, yMove) and moves the center of the Circle parameter by adding the xMove and yMove values to the coordinates of the center of the Circle parameter.
Circle III

static void moveCenter(Circle c, int xMove, int yMove) {
    c.xCenter += xMove;
    c.yCenter += yMove;
}

Copyright © 2012 Pearson Education, Inc.
Consider the following method of the class Circle

```java
static void moveCreateCircle(Circle c, int xM, int yM) {
    c = new Circle(c.xCenter + xM, c.yCenter + yM, c.radius);
}
```

What is printed?

```java
Circle c = new Circle(1, 1, 4);
moveCreateCircle(c, 3, 2);
System.out.println(c);
```
Consider the following method of the class Circle

```java
static void moveCreateCircle(Circle c, int xM, int yM) {
    c = new Circle(c.xCenter + xM, c.yCenter + yM, c.radius);
}
```

What is printed?

```java
Circle c = new Circle(1, 1, 4);
moveCreateCircle(c, 3, 2);
System.out.println(c);
```

Circle: x = 1; y = 1; r = 4
Method Overloading

• Let's look at one more important method design issue: **method overloading**

• *Method overloading* is the process of **giving a single method name multiple definitions in a class**

• *If a method is overloaded, the method name is not sufficient to determine which method is being called*

• The **signature of a method** is: the number, type, and **order** of the parameters

• The **signature** of each overloaded method **must be unique**
Method Overloading

• The compiler determines which method is being invoked by analyzing the parameters

```java
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}
```

Invocation

```java
result = tryMe(25, 4.32)
```
Method Overloading

• The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

• The following lines invoke different versions of the `println` method:

```
System.out.println("The total is:");
System.out.println(total);
```
Overloading Methods

• The return type of the method is **not** part of the signature

• That is, *overloaded methods cannot differ only by their return type*

• Constructors can be overloaded

• Overloaded constructors provide multiple ways to initialize a new object
Quiz

• For each of the following pairs of method headers, state whether or not the signatures are distinct

a. `String describe (String name, int count)`
   `String describe (int count, String name)`

b. `void count()`
   `int count()`

c. `int howMany (int comparableValue)`
   `int howMany (int ceiling)`

d. `boolean greater (int value1)`
   `boolean greater (int value1, int value2)`
Quiz

• For each of the following pairs of method headers, state whether or not the signatures are distinct

a. `String describe (String name, int count)`
   `String describe (int count, String name)`  **YES**

b. `void count()`
   `int count()`  **NO**

c. `int howMany (int comparableValue)`
   `int howMany (int ceiling)`  **NO**

d. `boolean greater (int value1)`
   `boolean greater (int value1, int value2)`  **YES**
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
Testing

• Testing can mean many different things

• It certainly includes **running** a completed program with various inputs

• It also includes any **evaluation** performed by **human** or computer to **assess quality**

• Some evaluations should occur before coding even begins

• The earlier we find an problem, the easier and cheaper it is to fix
Testing

• The goal of testing is to find errors

• As we find and fix errors, we raise our confidence that a program will perform as intended

• We can never really be sure that all errors have been eliminated

• So when do we stop testing?
  – Conceptual answer: Never
  – Cynical answer: When we run out of time
  – Better answer: When we are willing to risk that an undiscovered error still exists
Reviews

• A review is a meeting in which several people examine a design document or section of code

• It is a common and effective form of human-based testing

• Presenting a design or code to others:
  – makes us think more carefully about it
  – provides an outside perspective

• Reviews are sometimes called inspections or walkthroughs
Test Cases

- A test case is a set of input and user actions, coupled with the expected results.
- Often test cases are organized formally into test suites which are stored and reused as needed.
- For medium and large systems, testing must be a carefully managed process.
- Many organizations have a separate Quality Assurance (QA) department to lead testing efforts.
Defect and Regression Testing

- **Defect testing** is the execution of test cases to uncover errors.
- The act of fixing an error may introduce new errors.
- After fixing a set of errors we should perform **regression testing** – running previous test suites to ensure new errors haven't been introduced.
- It is not possible to create test cases for all possible input and user actions.
- Therefore we should design tests to maximize their ability to find problems.
Black-Box Testing

• In *black-box testing*, test cases are developed without considering the internal logic

• They are based on the input and expected output

• Input can be organized into *equivalence categories*

• Two input values in the same equivalence category *should* produce similar results

• Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories
Quiz

• Go back to the RationalTester program
• Is this a satisfactory example of black-box testing?
• What else would you propose to do?
White-Box Testing

- *White-box testing* focuses on the internal structure of the code
- The goal is to ensure that every path through the code is tested
- Paths through the code are governed by any conditional or looping statements in a program
- A good testing effort will include both black-box and white-box tests
Quiz

• Go back in these slides to the **RationalNumber class** and the **RationalTester program**

• Is this a satisfactory example of **white-box testing**?

• What else would you propose to do?
Quiz

Of the various phases in software development, which of the following is usually the lengthiest?

A) specification
B) design
C) implementation
D) testing
E) maintenance
Of the various phases in software development, which of the following is usually the lengthiest?

A) specification
B) design
C) implementation
D) testing
E) maintenance

Software requires modification (such as new requirements such as new features or I/O specifications) and so the maintenance phase is on-going whereas the other phases end once the software has been released and is in use.
Quiz

Modifying a program in order to eliminate deficiencies is done in the ________ phase of the development cycle.
A) design
B) implementation
C) testing
D) use
E) maintenance
Quiz

Modifying a program in order to eliminate deficiencies is done in the __________ phase of the development cycle.
A) design
B) implementation
C) testing
D) use
E) maintenance

Testing is used to find errors. Deficiencies are more commonly identified by the users of the system once the system has been released.
Outline

Software Development Activities
Identifying Classes and Objects
Static Variables and Methods
Class Relationships
Interfaces
Enumerated Types Revisited
Method Design
Testing
GUI Design and Layout
GUI Design

• We must remember that the goal of software is to help the user solve the problem

• To that end, the GUI designer should:
  – Know the user
  – Prevent user errors
  – Optimize user abilities
  – Be consistent

• Let's discuss each of these in more detail
Do interfaces matter?

• Play this two-Person Game:
  • Every player can take the numbers 1, 2, 3, ..., 9
  • Alternate turns, taking one number at a time (if a number is taken by your opponent you cannot take it)
  • A player wins with any 3 numbers that sum to 15
    • 1, 3, 9, 5 wins because 1+9+5 equals 15
    • 9, 6, 8, 7 does not win, because no 3 of them sum to 15
  • Tie if numbers are all used up without a winner

Adapted from Norman, "Design of everyday things"
Magic square

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>
Users
Know the User

• Knowing the user implies an understanding of:
  – the user's true needs
  – the user's common activities
  – the user's level of expertise in the problem domain and in computer processing

• We should also realize these issues may differ for different users

• Remember, to the user, the interface is the program
Camera feature search

Our new camera feature search is the perfect way to find the digital cameras that best suit your specific requirements. Start your search by clicking on one or more of the basic body type icons. You can then narrow down your selection by adding search criteria from the pop-up menu. Add as many search criteria as you wish, but remember that the more specific you get, the fewer matches you’ll see. You can compare up to 20 cameras in more detail by adding them to your shortlist - just click the compare button next to any camera in the list. To find out more about choosing the best digital SLR for you, read our beginner’s guide.

- **Body style (63 cameras)**
  - Fixed lens cameras (515)
  - Interchangeable lens cameras (133)

  - Ultra compact (339)
  - SLR like (bridge) (54)
  - Large sensor Compact (12)
  - Rangefinders (11)
  - Compact SLR (35)
  - Mid-size SLR (28)
  - Large SLR (12)

Zoom range: Any

Show advanced search filters

63 matches found
Prevent User Errors

• Whenever possible, we should design user interfaces that **minimize possible user mistakes**

• We should choose the best GUI components for each task

• For example, in a situation where there are only a few valid options, using a menu or radio buttons would be better than an open text field

• Error messages should **guide** the user appropriately

*Important for your exam project*
Optimize User Abilities

• Not all users are alike – some may be more familiar with the system than others

• Knowledgeable users are sometimes called power users

• We should provide multiple ways to accomplish a task whenever reasonable
  – "wizards" to walk a user through a process
  – short cuts for power users

• Help facilities should be available but not intrusive
Wizard: My Product Advisor

The system decides what the wizard says

Possible user's requests

Now you can:
- Answer more questions that are important to you.
- See recommended cameras based on your preferences so far.
- Review what you have done or start over.

The system decides what the wizard says
Be Consistent

- **Consistency** is important – users get used to things appearing and working in certain ways

- **Colors** should be used *consistently* to indicate similar types of information or processing

- **Screen layout** should be consistent from one part of a system to another

- For example, error messages should appear in consistent locations
Layout Managers

• *A layout manager* is an object that determines the way that components are arranged in a container.

• There are several predefined layout managers defined in the Java API:

```
FlowLayout
Border Layout
Card Layout
Grid Layout
GridBag Layout
Box Layout
Overlay Layout
```

- Defined in the AWT
- Defined in Swing
### Predefined layout managers

<table>
<thead>
<tr>
<th>Layout Manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Border Layout</td>
<td>Organizes components into five areas (North, South, East, West and Center).</td>
</tr>
<tr>
<td>Box Layout</td>
<td>Organizes components into a single row or column.</td>
</tr>
<tr>
<td>Card Layout</td>
<td>Organizes components into one area such that only one is visible at any time.</td>
</tr>
<tr>
<td>Flow Layout</td>
<td>Organizes components from left to right, starting new rows as necessary.</td>
</tr>
<tr>
<td>Grid Layout</td>
<td>Organizes components into a grid of rows and columns.</td>
</tr>
<tr>
<td>GridBag Layout</td>
<td>Organizes components into a grid of cells, allowing components to span more than one cell.</td>
</tr>
</tbody>
</table>
Layout Managers

• Every **container** has a **default** layout manager, but we can explicitly set the layout manager as well

• Each layout manager has its own particular rules governing how the components will be arranged

• Some layout managers pay attention to a component's preferred size or alignment, while others do not

• A layout manager adjusts the layout as components are added and as containers are resized
Layout Managers

• We can use the `setLayout` method of a container to change its layout manager:

```java
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

• The following example uses a *tabbed pane*, a container which permits one of several panes to be selected.

• Each tabbed pane contains a panel that is controlled by a different layout manager.

• See `LayoutDemo.java`
• See `IntroPanel.java`
import javax.swing.*;

public class LayoutDemo {
    public static void main (String[] args) {
        JFrame frame = new JFrame ("Layout Manager Demo");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        continue
    }
}
continue

JTabbedPane tp = new JTabbedPane();
tp.addTab ("Intro", new IntroPanel());
tp.addTab ("Flow", new FlowPanel());
tp.addTab ("Border", new BorderPanel());
tp.addTab ("Grid", new GridPanel());
tp.addTab ("Box", new BoxPanel());

frame.getContentPane().add(tp);
frame.pack();
frame.setVisible(true);
import java.awt.*;
import javax.swing.*;

public class IntroPanel extends JPanel {
    // Sets up this panel with two labels.

    public IntroPanel() {
        setBackground (Color.green);

        JLabel l1 = new JLabel ("Layout Manager Demonstration");
        JLabel l2 = new JLabel ("Choose a tab to see an example of " + 
                                "a layout manager.");

        add (l1);
        add (l2);
    }
}
import java.awt.*;
import javax.swing.*;

public class IntroPanel extends JPanel {
    // Sets up this panel with two labels.
    public IntroPanel() {
        setBackground (Color.green);

        JLabel l1 = new JLabel ("Layout Manager Demonstration");
        JLabel l2 = new JLabel ("Choose a tab to see an example of a layout manager.");

        add (l1);
        add (l2);
    }
}
Flow Layout

- *Flow layout* puts as many components as possible on a row, then moves to the next row.

- Components are displayed in the order they are added to the container.

- Each row of components is centered horizontally by default, but could also be aligned left or right.

- The horizontal and vertical gaps between the components can be explicitly set (default is 5).

- See `FlowPanel.java`
import java.awt.*;
import javax.swing.*;

public class FlowPanel extends JPanel
{
    // -----------------------------------------------------------------
    // Sets up this panel with some buttons to show how flow layout
    // affects their position.
    // -----------------------------------------------------------------
    public FlowPanel()
    {
        setLayout (new FlowLayout());
        setBackground (Color.green);
    }
}

特别声明：未经授权，不得翻版、复制、转载、上传、下载或引用作商业用途。版权所有 © 2012 Pearson Education, Inc.
continue

    JButton b1 = new JButton ("BUTTON 1");
    JButton b2 = new JButton ("BUTTON 2");
    JButton b3 = new JButton ("BUTTON 3");
    JButton b4 = new JButton ("BUTTON 4");
    JButton b5 = new JButton ("BUTTON 5");

    add (b1);
    add (b2);
    add (b3);
    add (b4);
    add (b5);
    }
}
import java.awt.

public class LayoutManagerDemo {

    public static void main(String[] args) {
        javax.swing.JFrame frame = new javax.swing.JFrame("LayoutManager Demo");
        frame.setSize(400, 400);
        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);

        javax.swing.JButton b1 = new javax.swing.JButton("BUTTON 1");
        javax.swing.JButton b2 = new javax.swing.JButton("BUTTON 2");
        javax.swing.JButton b3 = new javax.swing.JButton("BUTTON 3");
        javax.swing.JButton b4 = new javax.swing.JButton("BUTTON 4");
        javax.swing.JButton b5 = new javax.swing.JButton("BUTTON 5");

        frame.add(b1);
        frame.add(b2);
        frame.add(b3);
        frame.add(b4);
        frame.add(b5);

        frame.setVisible(true);
    }
}
Border Layout

- A *border layout* defines five areas into which components can be added
Border Layout

• Each area displays one component (which could be a container such as a JPanel)

• Each of the four outer areas enlarges as needed to accommodate the component added to it

• If nothing is added to the outer areas, they take up no space and other areas expand to fill the void

• The center area expands to fill space as needed

• See BorderPanel.java
import java.awt.*;
import javax.swing.*;

public class BorderPanel extends JPanel
{
    // Sets up this panel with a button in each area of a border
    // layout to show how it affects their position, shape, and size.
    public BorderPanel()
    {
        setLayout (new BorderLayout());
        setBackground (Color.green);
    }
}
continue

    JButton b1 = new JButton ("BUTTON 1");
    JButton b2 = new JButton ("BUTTON 2");
    JButton b3 = new JButton ("BUTTON 3");
    JButton b4 = new JButton ("BUTTON 4");
    JButton b5 = new JButton ("BUTTON 5");

    add (b1, BorderLayout.CENTER);
    add (b2, BorderLayout.NORTH);
    add (b3, BorderLayout.SOUTH);
    add (b4, BorderLayout.EAST);
    add (b5, BorderLayout.WEST);
}
}
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1, BorderLayout.CENTER);
add(b2, BorderLayout.NORTH);
add(b3, BorderLayout.SOUTH);
add(b4, BorderLayout.EAST);
add(b5, BorderLayout.WEST);
}
Grid Layout

- A *grid layout* presents a container’s components in a rectangular grid of rows and columns.
- One component is placed in each cell of the grid, and *all cells have the same size*.
- Components fill the grid from left-to-right and top-to-bottom (by default).
- *The size of each cell is determined by the overall size of the container*.
- See `GridPanel.java`
import java.awt.*;
import javax.swing.*;

public class GridPanel extends JPanel {
    // Sets up this panel with some buttons to show how grid
    // layout affects their position, shape, and size.
    public GridPanel() {
        setLayout (new GridLayout (2, 3));
        setBackground (Color.green);
    }
}

// *****************************************
// GridPanel.java       Authors: Lewis/Loftus
//
// Represents the panel in the LayoutDemo program that demonstrates
// the grid layout manager.
// *****************************************
JButton b1 = new JButton ("BUTTON 1");
JButton b2 = new JButton ("BUTTON 2");
JButton b3 = new JButton ("BUTTON 3");
JButton b4 = new JButton ("BUTTON 4");
JButton b5 = new JButton ("BUTTON 5");

add (b1);
add (b2);
add (b3);
add (b4);
add (b5);
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1); add(b2); add(b3); add(b4); add(b5);
Box Layout

• A *box layout* organizes components *horizontally* (in one row) or *vertically* (in one column)

• Components are placed top-to-bottom or left-to-right in the order in which they are added to the container

• By **combining** multiple containers using box layout, many different configurations can be created

• *Multiple containers with box layouts are often preferred to one container that uses the more complicated gridbag layout manager*
Box Layout

• *Invisible components* can be added to a box layout container to take up space between components
  – *Rigid areas* have a fixed size
  – *Glue* specifies where excess space should go

• Invisible components are created using these methods of the `Box` class:
  
  ```java
  createRigidArea(Dimension d)
  createHorizontalGlue()
  createVerticalGlue()
  ```

• See `BoxPanel.java`
import java.awt.*;
import javax.swing.*;

public class BoxPanel extends JPanel {
    // Sets up this panel with some buttons to show how a vertical box layout (and invisible components) affects their position.
    public BoxPanel() {
        setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
        setBackground (Color.green);
    }
    continue
continue

    JButton b1 = new JButton("BUTTON 1");
    JButton b2 = new JButton("BUTTON 2");
    JButton b3 = new JButton("BUTTON 3");
    JButton b4 = new JButton("BUTTON 4");
    JButton b5 = new JButton("BUTTON 5");

    add (b1);
    add (Box.createRigidArea (new Dimension (0, 10)));
    add (b2);
    add (Box.createVerticalGlue());
    add (b3);
    add (b4);
    add (Box.createRigidArea (new Dimension (0, 20)));
    add (b5);
}
continue

```java
JButton b1 = new JButton("BUTTON 1");
JButton b2 = new JButton("BUTTON 2");
JButton b3 = new JButton("BUTTON 3");
JButton b4 = new JButton("BUTTON 4");
JButton b5 = new JButton("BUTTON 5");
add(b1);
add(Box.createRigidArea(new Dimension(0, 10)));
add(b2);
add(Box.createVerticalGlue());
add(b3);
add(b4);
add(Box.createRigidArea(new Dimension(0, 20)));
add(b5);
```
Borders

• A *border* can be put **around any Swing component** to define **how the edges** of the component **should be drawn**

• Borders can be used effectively to group components visually

• The *BorderFactory* class contains several static methods for creating border objects

• A border is applied to a component using the *setBorder* method
Borders

• An *empty border*
  – buffers the space around the edge of a component
  – otherwise has no visual effect

• A *line border*
  – surrounds the component with a simple line
  – the line's color and thickness can be specified

• An *etched border*
  – creates the effect of an etched groove around a component
  – uses colors for the highlight and shadow
Borders

• A bevel border
  – can be raised or lowered
  – uses colors for the outer and inner highlights and shadows

• A titled border
  – places a title on or around the border
  – the title can be oriented in many ways

• A matte border
  – specifies the sizes of the top, left, bottom, and right edges of the border separately
  – uses either a solid color or an image
Borders

• A *compound border*
  – is a combination of two borders
  – one or both of the borders can be a compound border

• See `BorderDemo.java`
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public class BorderDemo
{
    //---
    // Creates several bordered panels and displays them.
    //---
    public static void main (String[] args)
    {
        JFrame frame = new JFrame ("Border Demo");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout (new GridLayout (0, 2, 5, 10));
        panel.setBorder (BorderFactory.createEmptyBorder (8, 8, 8, 8));

        JPanel p1 = new JPanel();
        p1.setBorder (BorderFactory.createLineBorder (Color.red, 3));
        p1.add (new JLabel ("Line Border"));
        panel.add (p1);
    }
}
continue

```java
JPanel p2 = new JPanel();
p2.setBorder (BorderFactory.createEtchedBorder ());
p2.add (new JLabel ("Etched Border"));
panel.add (p2);

JPanel p3 = new JPanel();
p3.setBorder (BorderFactory.createRaisedBevelBorder ());
p3.add (new JLabel ("Raised Bevel Border"));
panel.add (p3);

JPanel p4 = new JPanel();
p4.setBorder (BorderFactory.createLoweredBevelBorder ());
p4.add (new JLabel ("Lowered Bevel Border"));
panel.add (p4);

JPanel p5 = new JPanel();
p5.setBorder (BorderFactory.createTitledBorder ("Title");
p5.add (new JLabel ("Titled Border"));
panel.add (p5);

JPanel p6 = new JPanel();
TitledBorder tb = BorderFactory.createTitledBorder ("Title");
tb.setTitleJustification (TitledBorder.RIGHT);
p6.setBorder (tb);
p6.add (new JLabel ("Titled Border (right)"));
panel.add (p6);
```
continue
continue

    JPanel p7 = new JPanel();
    Border b1 = BorderFactory.createLineBorder (Color.blue, 2);
    Border b2 = BorderFactory.createEtchedBorder ();
    p7.setBorder (BorderFactory.createCompoundBorder (b1, b2));
    p7.add (new JLabel ("Compound Border");
    panel.add (p7);

    JPanel p8 = new JPanel();
    Border mb = BorderFactory.createMatteBorder (1, 5, 1, 1, Color.red);
    p8.setBorder (mb);
    p8.add (new JLabel ("Matte Border");
    panel.add (p8);

    frame.getContentPane().add (panel);
    frame.pack();
    frame.setVisible(true);
}
continue

JPanel p7 = new JPanel();
Border b1 = BorderFactory.createLineBorder(Color.blue, 2);
Border b2 = BorderFactory.createEtchedBorder();
p7.setBorder(BorderFactory.createCompoundBorder(b1, b2));
p7.add(new JLabel("Compound Border"));
panel.add(p7);

JPanel p8 = new JPanel();
Border mb = BorderFactory.createMatteBorder(1, 5, 1, 1, Color.red);
p8.setBorder(mb);
p8.add(new JLabel("Matte Border"));
panel.add(p8);
frame.getContentPane().add(panel);
frame.pack();
frame.setVisible(true);
Summary

• Chapter 7 has focused on:
  – software development activities
  – determining the classes and objects that are needed for a program
  – the relationships that can exist among classes
  – the static modifier
  – writing interfaces
  – the design of enumerated type classes
  – method design and method overloading
  – GUI design and layout managers
Quiz

Assume a class `Foo` implements `Comparable`. Without knowing anything else about the `Foo` class, write an `equals` method that returns true if the `Foo parameter` passed to the `equals` method is equal to this `Foo` as determined by using the implementation of `Comparable`.
Quiz

Assume a class `Foo` implements `Comparable`. Without knowing anything else about the `Foo` class, write an `equals` method that returns true if the `Foo` parameter passed to the `equals` method is equal to this `Foo` as determined by using the implementation of `Comparable`.

```java
public boolean equals(Foo a) {
    if(compareTo(a) == 0)
        return true;
    else
        return false;
}
```
Exercises

• Write a method called average that accepts two integer parameters and returns their average as a floating point value.

• Overload the average method of the previous exercise such that if three integers are provided as parameters, the method returns the average of all three.

• Explain why a static method cannot refer to an instance variable.

• Create an interface called VCR that has methods that represent the standard operations on a video cassette recorder (play, stop, etc.). Define the method signatures any way you desire. Describe how a class might implement this interface.
public double average(int num1, int num2) {
    return (num1 + num2) / 2.0;
}

public double average(int num1, int num2, int num3) {
    return (num1 + num2 + num3) / 3.0;
}
Solutions

• A static method is invoked through a class rather than through an object of the class. No object of the class needs to be instantiated in order to invoke a static method. If no object is instantiated, no instance variable exists. Hence, a static method cannot refer to an instance variable.
public interface VCR {
    public String play();
    public String stop();
    public String record(int start, int end);
    public String pause();
}

• A class implementing VCR would include an implements clause in the class header, such as:

    public class MyVCR implements VCR

• The class would contain, among other things, four methods with signatures that match those specified in the interface.