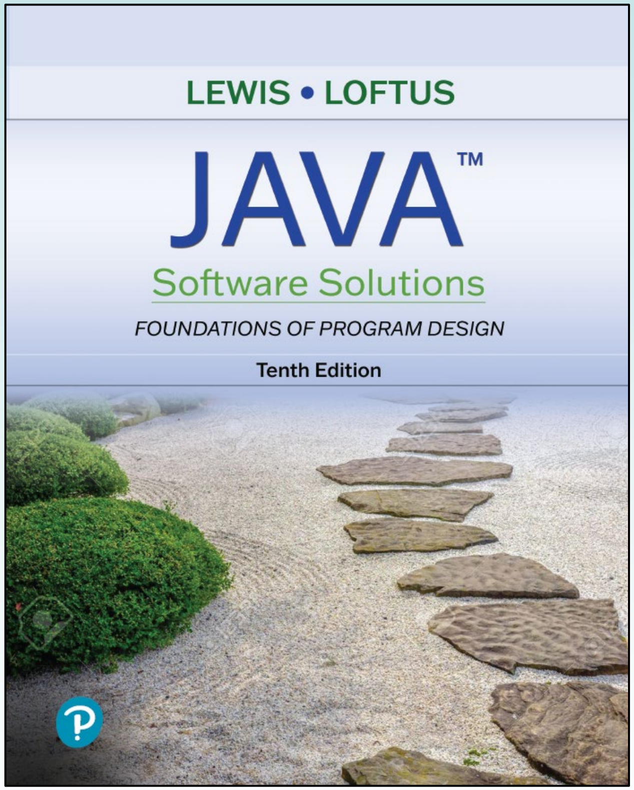
Chapter 7
Object-Oriented Design



Java Software Solutions
Foundations of Program Design
10th Edition

(edits: 05/01/2026)

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Object-Oriented Design

- 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

§7.1 Program Development

The creation of software involves four activities:

1. establishing the requirements
2. designing
3. implementation (writing code)
4. testing

These activities are not strictly linear – they overlap and interact and may be repeated.

1. Requirements

- *Software requirements* specify the tasks that a program must accomplish
  - <u>what</u> you want, not how to do it
  - requirements should be specified before coding begins
- Often an initial set of requirements is provided, but they should be critiqued and expanded
  - 50% of software projects fail
  - 50% of small businesses fail in their first year

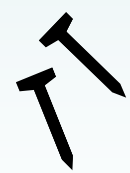
- It is difficult to establish detailed, unambiguous, and complete requirements
  - Often requirements change as the project develops
  - Often a complete prototype is written only to discover what the requirements should have been
    - “prototyping languages”
    - after studying the prototype, the real program is written
- Careful attention to the requirements can save significant time and expense in the overall project

2. 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 describes how individual methods will accomplish their tasks
  - Possibly done with language-independent tools (sometimes flowcharts, sometimes UML diagrams)

3. Implementation

- *Implementation* is the process of translating a design into source code
  - Coding might be as little as 10% of the time spent on a software project !!!
- Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
  - "architecture" vs. "hammering nails"



3. Implementation

- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation
  - ideally, coding should be nearly "mechanical"
  - entry-level jobs are often coding (or testing)
  - top-level jobs are analysis and design

4. Testing

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

- *Debugging* is the process of determining the cause of a problem and fixing it
  - Testing finds a problem, debugging fixes it

4. Testing

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

- various data sets used:
  - typical, random, extreme cases
  - look at the "boundaries"
  - check for off-by-one errors
- testers are often not the programmers who wrote the code

Fred Brooks

- “The … question … is not *whether* to build a pilot system and throw it away. You *will* do that. Hence *plan to throw one away; you will, anyhow.”*
- “Well over half of the time you spend working on a project (on the order of 70 percent) is spent thinking…”
- http://en.wikiquote.org/wiki/Fred_Brooks

Fred Brooks (ca. 1965)

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 might be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the real-world 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 A.

- Of course, not all nouns will correspond to a class or object in the final solution

☺ Identifying Classes and Objects

- A class represents a group of objects with the same behavior

- 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

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

§7.3 Static Class Members

- A static method is invoked through the name of the class it is in
  - no object needed
  - the method is part of the class, not part of an object
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25);
```

```java
import java.util.Scanner;
public class StaticDemo
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner( System.in );
        System.out.print("Enter a num: ");
        double num = scan.nextDouble();
        double root = Math.sqrt( num );
        System.out.println("Square Root is " + root );
    }
}
```

`sqrt()` is a method that is part of the `Math` class.
Don't need to make a Math object to run it.

The static Modifier

- A static variable is part of its class, not part of any object
- Declare static methods and variables using the **`static`** modifier
  - It puts the method or variable in the class not in objects of that class
- Static methods are sometimes called *class methods*
- static variables are sometimes called *class variables*

Static Variables

- Each object has its own instance variables. But if a variable is declared as static, only one copy of the variable exists

```
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
- If one object changes a class variable, all other objects of that class see the change

Static Methods

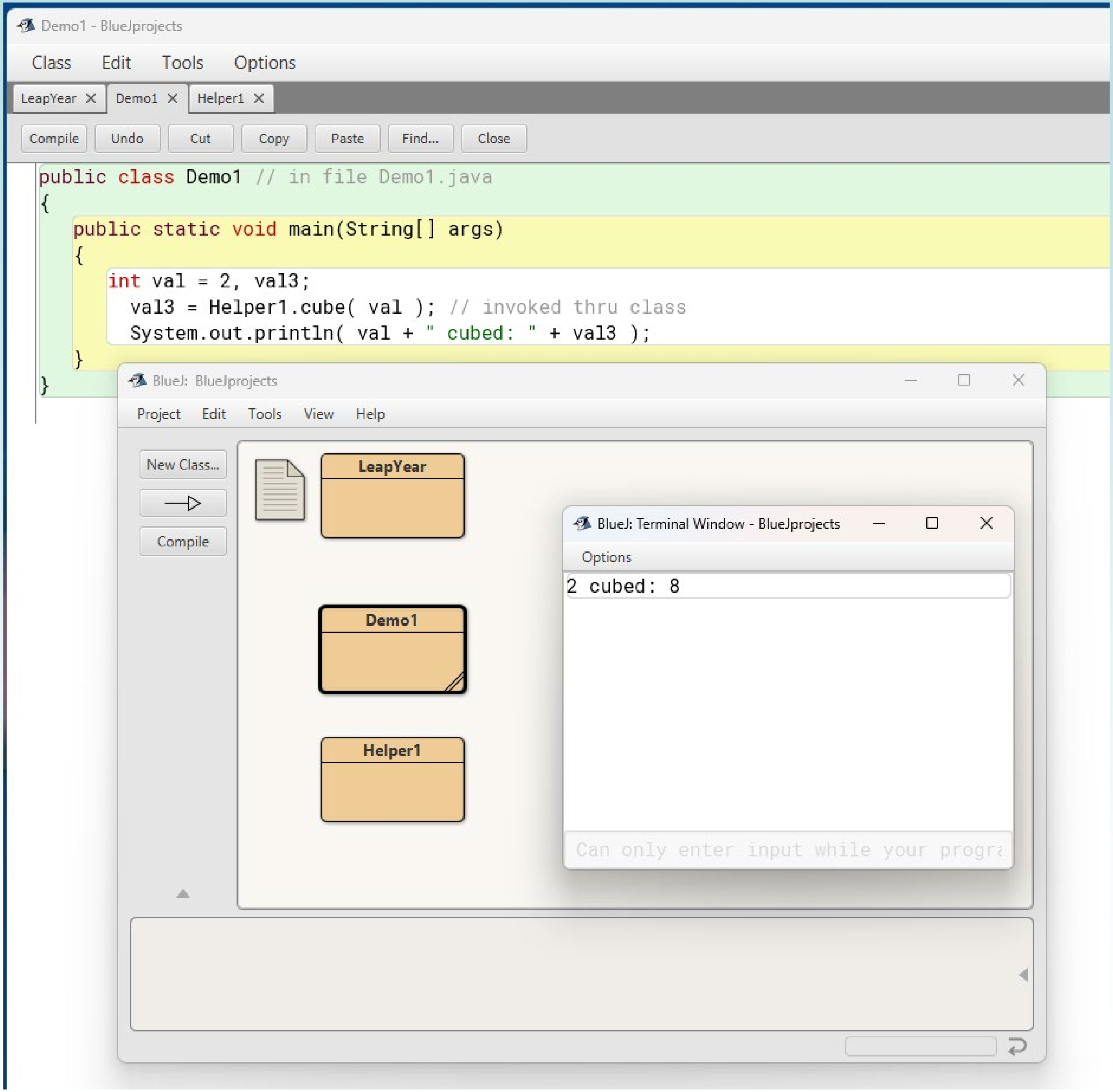
```
public class Helper1
{
   public static int cube(int num)
   {
      return num * num * num;
   }
}
```

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

```
value = Helper1.cube(4);
```

Try with BlueJ

```
public class Helper1 // in file Helper1.java
{
   public static int cube(int num)
   {
      return num * num * num;
   }
}

public class Demo1 // in file Demo1.java
{
   public static void main(String[] args)
   {
      int val = 2, val3;
      val3 = Helper1.cube( val ); // invoked thru class
      System.out.println( val + " cubed: " + val3 );
   }
}
```


Demo1 - BlueJprojects
Class
Edit
Tools
Options
LeapYear
Demo1
Helper1
Compile
Undo
Cut
Copy
Paste
Find...
Close
public class Demo1 // in file Demo1.java
{
public static void main(String[] args)
{
int val = 2, val3;
val3 = Helper1.cube(val); // invoked thru class
System.out.println(val + " cubed: " + val3);
}
}
BlueJ: BlueJprojects
Project
Edit
Tools
View
Help
New Class...
Compile
LeapYear
Demo1
Helper1
BlueJ: Terminal Window - BlueJprojects
Options
2 cubed: 8
Can only enter input while your progr

Select C:\Windows\System32\cmd.exe

```
C:\Users\kjell\OneDrive - CCSU\AA151\Static>javac Demo1.java Helper1.java

C:\Users\kjell\OneDrive - CCSU\AA151\Static>java Demo1
2 cubed: 8

C:\Users\kjell\OneDrive - CCSU\AA151\Static>_
```

C:\Windows\System32\cmd.exe

```
C:\Users\kjell\OneDrive - CCSU\AA151\Static>javac Demo2.java Helper2.java

C:\Users\kjell\OneDrive - CCSU\AA151\Static>java Demo2
2 cubed: 8

C:\Users\kjell\OneDrive - CCSU\AA151\Static>
```

C:\Windows\System32\cmd.exe

```
C:\Users\kjell\OneDrive - CCSU\AA151\Static>javac Demo3.java Helper3.java

C:\Users\kjell\OneDrive - CCSU\AA151\Static>java Demo3
2 cubed: 8

C:\Users\kjell\OneDrive - CCSU\AA151\Static>
```

```java
public class Helper2 // in file Helper2.java
{
   public int cube(int num)  // not static
   {
      return num * num * num;
   }
}

public class Demo2 // in file Demo2.java
{
   public static void main(String[] args)
   {
      int val = 2, val3;
      Helper2 help = new Helper2(); // make an object
      val3 = help.cube( val );  // use object's method
      System.out.println( val + " cubed: " + val3 );
   }
}
```

Static Class Members

- The order of the modifiers `public static` can be reversed, `static public` 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 use instance variables of any object because instance variables don't exist until an object exists
- However, a static method can reference static variables in its class or its own local variables

```java
public class Helper3  // in file Helper3.java
{
   static int temp;
   public static int cube(int num)
   {
      int x = num ;  // static method can access
      temp = x * x;  // static variables and its
                     // own local variables
      return temp * x;
   }
}

public class Demo3  // in file Demo.java
{
   public static void main(String[] args)
   {
       int val = 2, val3;
       val3 = Helper3.cube( val );
       System.out.println( val + " cubed: " + val3 );
   }
}
```

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`

```java
//********************************************************************
//  SloganCounter.java       Author: Lewis/Loftus
//********************************************************************
public class SloganCounter
{
   //-----------------------------------------------------------------
   //  Creates several Slogan objects and prints the number of
   //  objects that were created.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Slogan obj; // this will point at different Slogans as the program runs

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

      obj = new Slogan("Don't Worry. Be Happy."); // Alamo is now garbage
      System.out.println(obj);
```

Uses toString() method

```java
      obj = new Slogan("Live Free or Die.");  // Happy is now garbage
      System.out.println(obj);

      obj = new Slogan("Talk is Cheap.");     // Live is now garbage
      System.out.println(obj);
      System.out.println();
      System.out.println("Slogans created: " + Slogan.getCount());
   }
}
```

Output

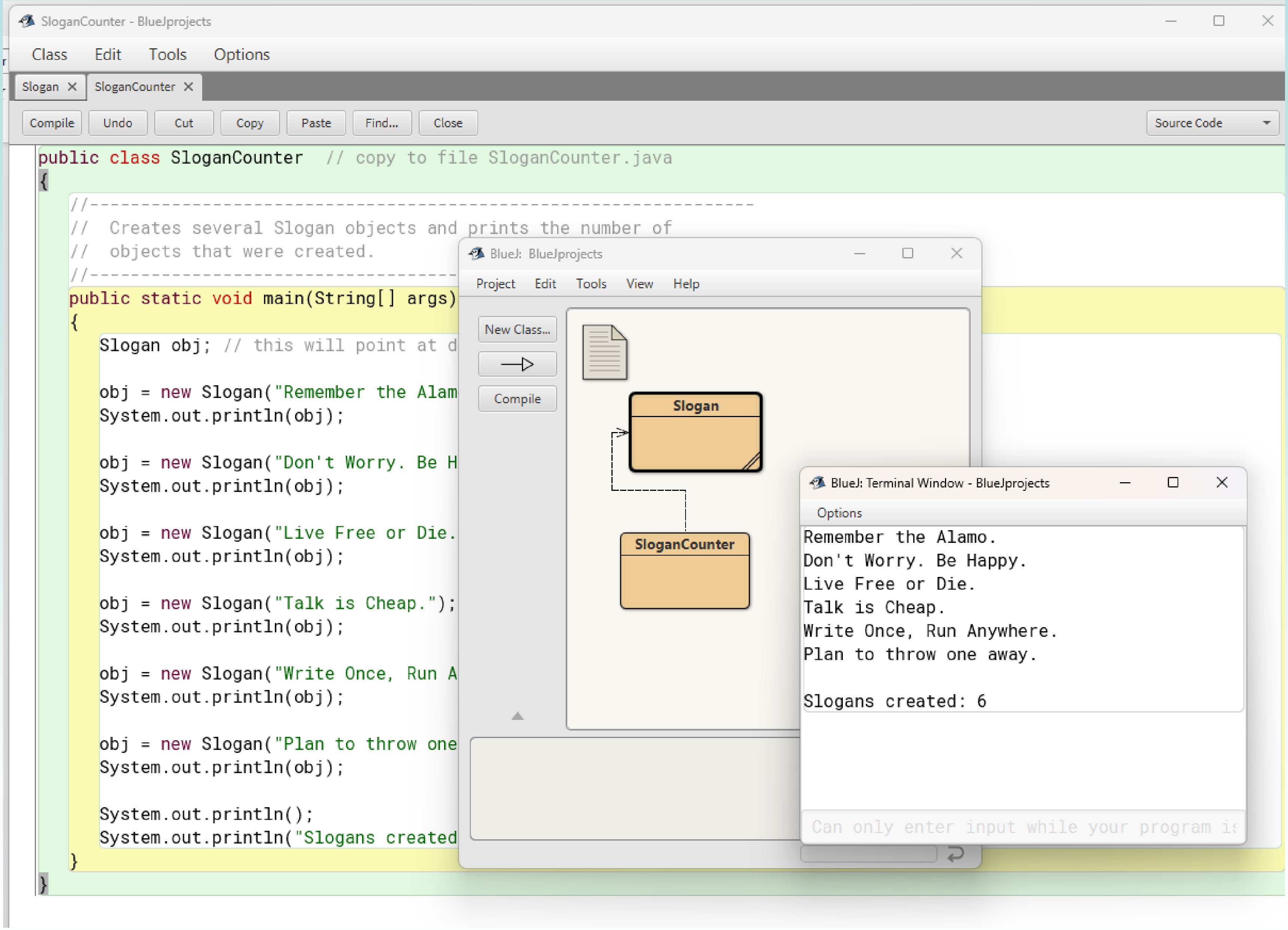
```
Remember the Alamo.
Don't Worry. Be Happy.
Live Free or Die.
Talk is Cheap.

Slogans created: 4
```

```java
//********************************************************************
//  Slogan.java       Author: Lewis/Loftus
//
//  Represents a single slogan string.
//********************************************************************

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++;
   }
```

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

   //-----------------------------------------------------------------
   //  Returns the number of instances of this class that have been
   //  created, even if many of them are now garbage.  The static
   //  variable is part of the class, and exists as long as the
   //  program runs.
   //-----------------------------------------------------------------
   public static int getCount()
   {
      return count;
   }
}
```

```java
public class Slogan  // copy to file Slogan.java
{
   private String phrase;          // each object has a phrase
   private static int count = 0;  // count is part of the class,
                                  // just one of these

   public Slogan(String str)
   {
      phrase = str;
      count++;
   }

   public String toString()
   {
      return phrase;
   }

  public static int getCount()
  {
      return count;
   }
}
```

```java
public class SloganCounter  // copy to file SloganCounter.java
{
   //-----------------------------------------------------------------
   //  Creates several Slogan objects and prints the number of
   //  objects that were created.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Slogan obj; // this will point at different Slogans as the program runs

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

      obj = new Slogan("Don't Worry. Be Happy."); // Alamo is now garbage
      System.out.println(obj);

      obj = new Slogan("Live Free or Die.");  // Happy is now garbage
      System.out.println(obj);

      obj = new Slogan("Talk is Cheap.");     // Live is now garbage
      System.out.println(obj);

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

      obj = new Slogan("Plan to throw one away.");  // Write is garbage
      System.out.println(obj);

      System.out.println();
      System.out.println("Slogans created: " + Slogan.getCount());
   }
}
```


SloganCounter - BlueJprojects
Class
Edit
Tools
Options
Slogan
SloganCounter
Compile
Undo
Cut
Copy
Paste
Find...
Close
Source Code
public class SloganCounter // copy to file SloganCounter.java
{
//---
// Creates several Slogan objects and prints the number of
// objects that were created.
//-----------------------------------
public static void main(String[] args)
{
Slogan obj; // this will point at d
obj = new Slogan("Remember the Alam
System.out.println(obj);
obj = new Slogan("Don't Worry. Be H
System.out.println(obj);
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 A
System.out.println(obj);
obj = new Slogan("Plan to throw one
System.out.println(obj);
System.out.println();
System.out.println("Slogans created
}
}
BlueJ: BlueJprojects
Project
Edit
Tools
View
Help
New Class...
Compile
Slogan
SloganCounter
BlueJ: Terminal Window - BlueJprojects
Options
Remember the Alamo.
Don't Worry. Be Happy.
Live Free or Die.
Talk is Cheap.
Write Once, Run Anywhere.
Plan to throw one away.
Slogans created: 6
Can only enter input while your program is

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)

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

§7.4 Class Relationships

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

- Three of the most common relationships:
  - Dependency: A ***uses*** $\dashrightarrow$ B
  - Aggregation: A ***has-a*** ——◆ B
  - Inheritance: A ***is-a*** $\longrightarrow$ B

- Inheritance is discussed in Chapter 9

Dependency

- A *dependency* exists when one class uses another in some way, usually by invoking the methods of the other
- 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 reference of the same class as a parameter
- For example, the `concat` method of the `String` class takes as a parameter another `String` object

```
String str1 = "Don't worry";
String str2 = "Be Happy";
String str3 = str1.concat(str2);
```

Dependency

- Example: `RationalNumber`
- A rational number is the ratio of two integers
- Several methods of the `RationalNumber` class accept another `RationalNumber` object reference as a parameter
- See `RationalTester.java`
- See `RationalNumber.java`

```java
//********************************************************************
//  RationalTester.java       Author: Lewis/Loftus
//
//  Driver to exercise the use of multiple Rational objects.
//********************************************************************

public class RationalTester
{
   //-----------------------------------------------------------------
   //  Creates some rational number objects and performs various
   //  operations on them.
   //-----------------------------------------------------------------
   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);
```

Uses toString() method

```java
    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);
  }
}
```

The methods of r1 create new RationalNumber objects using information in r1 and r2. Neither r1 nor r2 are changed.

Output

```
First rational number: 3/4
Second rational number: 1/3
r1 and r2 are NOT equal.
The reciprocal of r1 is: 4/3
r1 + r2: 13/12
r1 - r2: 5/12
r1 * r2: 1/4
r1 / r2: 9/4
```

```
//********************************************************************
//  RationalNumber.java       Author: Lewis/Loftus
//
//  Represents one rational number with a numerator and denominator.
//********************************************************************

public class RationalNumber
{
   private int numerator, denominator;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the rational number by ensuring a nonzero
   //  denominator and making only the numerator signed.
   //-----------------------------------------------------------------
   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;
      }
```

Instance variables. Every RationalNumber has these.

Parameters. These are values the constructor is given. It uses them to initialize the instance variables.

```java
        numerator   = numer;
        denominator = denom;

        reduce(); // reduce rational number to lowest terms
    }
```

Call this method within the object being constructed to alter its instance variables.

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

    //-----------------------------------------------------------------
    //  Returns the denominator of this rational number.
    //-----------------------------------------------------------------
    public int getDenominator()
    {
        return denominator;
    }
```

```java
//-----------------------------------------------------------------
//  Returns the reciprocal of this rational number.
//  This method constructs another RationalNumber and
//  returns a reference to it. So do the following four methods.
//-----------------------------------------------------------------
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.  3/4 + 1/3 = (3*3 + 1*4)/ (4*3) = 13/12
//-----------------------------------------------------------------
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);
}
```

```
//-----------------------------------------------------------------
//  Subtracts the rational number passed as a parameter from this
//  rational number.
//-----------------------------------------------------------------
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);
}

//-----------------------------------------------------------------
//  Multiplies this rational number by the one passed as a
//  parameter. (2/3) * (5/7) = (2*5)/(3*7)
//-----------------------------------------------------------------
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() );
}
```

```java
//-----------------------------------------------------------------
//  Returns this rational number as a string.
//-----------------------------------------------------------------
public String toString()
{
   String result;
   if (numerator == 0)
      result = "0";
   else
      if (denominator == 1)
         result = numerator + "";
      else
         result = numerator + "/" + denominator;
   return result;
}
```

```java
//-----------------------------------------------------------------
//  Reduces this rational number by dividing both the numerator
//  and the denominator by their greatest common divisor
//  (which might be 1).
//-----------------------------------------------------------------
private void reduce()
{
   if (numerator != 0)
   {
      int common = gcd( Math.abs(numerator), denominator );

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

6/8

(2*3) / (2*4)   common = 2

3/4

```java
   //-----------------------------------------------------------------
   //  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;
   }
}
```

Euclid's Algorithm

- Finds the Greatest Common Divisor of two integers
- GCD(35, 15) == 5
- If two integers have a common divisor, the difference of the two integers has the same divisor.
- To find the common divisor, keep subtracting the smaller integer from the larger integer until a common value is reached.
- The common value will be 1 if the integers are *relatively prime.*

Euclid's Algorithm

Say x and y have a common divisor, say d.

- Then $x = md$ and $y = nd$
- Then $(x - y) = md - nd = (m - n)d = kd$
- So the difference kd has the same divisor d

Say x is 35 and y is 15

35 and 15:	subtract smaller from larger: 20
	35, 15, and 20 all have the same divisor
	replace the larger with the result
20 and 15:	subtract smaller from larger: 5
	20, 15, and 5 all have the same divisor
	replace the larger with the result
5 and 15:	subtract smaller from larger: 10
	15, 10, and 5 all have the same divisor
	replace the larger with the result
5 and 10:	subtract smaller from larger: 5
	10, 5, and 5 all have the same divisor
	replace the larger with the result
5 and 5:	are equal, so: gcd = 5

Say x is 23 and y is 8

23 and 8:	subtract smaller from larger: 15 replace the larger with the result
15 and 8:	subtract smaller from larger: 7 replace the larger with the result
8 and 7:	subtract smaller from larger: 1 replace the larger with the result
7 and 1:	subtract smaller from larger: 6 replace the larger with the result
6 and 1:	subtract smaller from larger: 5 replace the larger with the result

5 and 1:	subtract smaller from larger: 4 replace the larger with the result
4 and 1:	subtract smaller from larger: 3 replace the larger with the result
3 and 1:	subtract smaller from larger: 2 replace the larger with the result
2 and 1:	subtract smaller from larger: 1 replace the larger with the result
1 and 1:	are equal, so: gcd = 1

Say x is 51 and y is 39

51 and 39: subtract smaller from larger: 12
replace larger with the result

39 and 12: subtract smaller from larger: 27
replace larger with the result

27 and 12: subtract smaller from larger: 15
replace larger with the result

15 and 12: subtract smaller from larger: 3
replace larger with the result

3 and 12: subtract smaller from larger: 9
replace larger with the result

3 and 9: subtract smaller from larger: 6
replace larger with the result

3 and 6: subtract smaller from larger: 3
replace larger with the result

3 and 3: are equal, so: gcd = 3

Aggregation

- An *aggregate* is an object that is made up of other objects (eg Slogan, p. 35)
- Therefore aggregation is a *has-a* relationship
  - A car *has-a* chassis
- An aggregate object contains references to other objects as its instance data
  - Often people speak as if one object were "made of" several other objects.

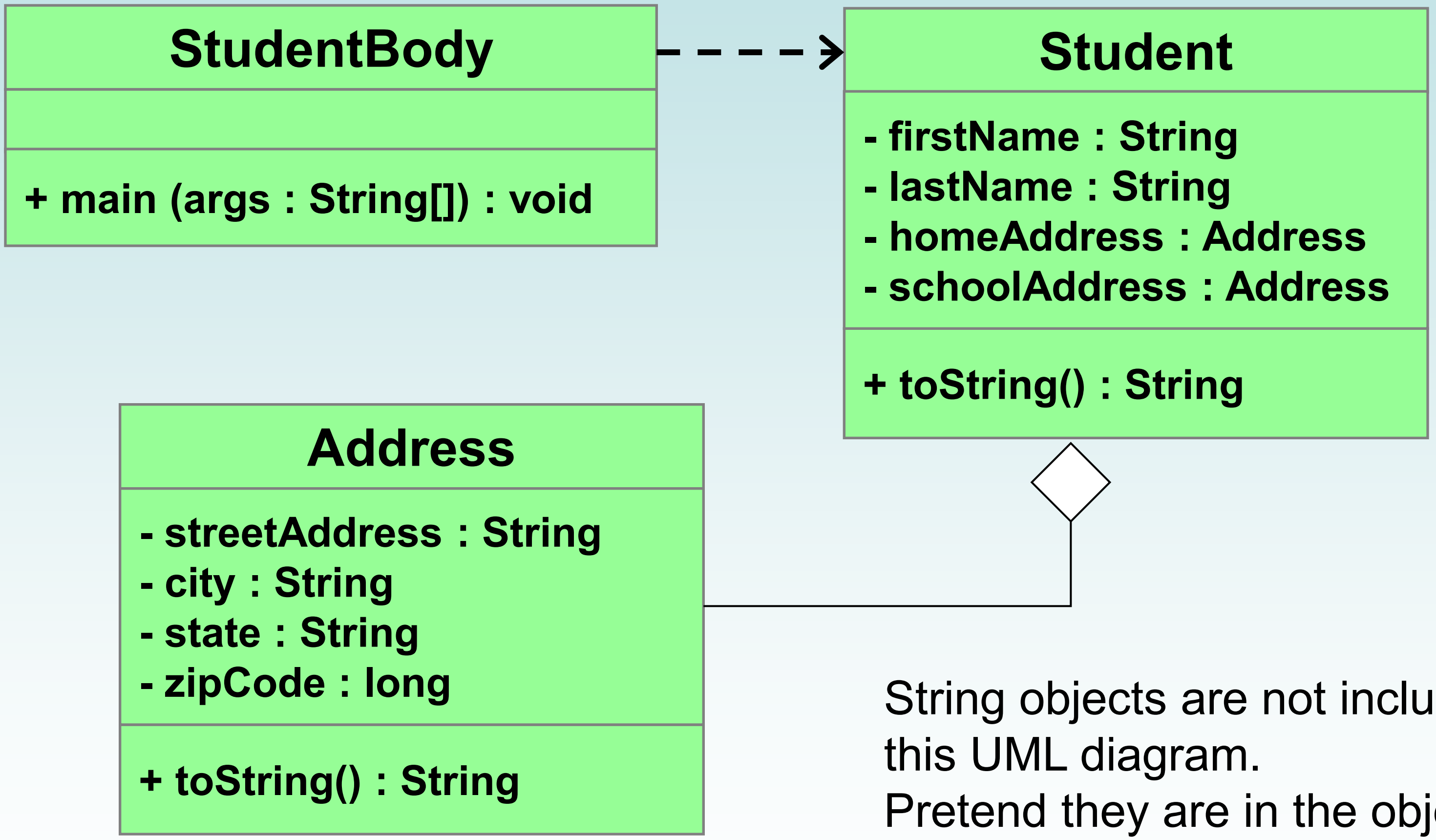
  - But in Java, aggregate objects contain <u>references</u> to other objects.

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)
- Aggregation: A 

B
- See `StudentBody.java`
- See `Student.java`
- See `Address.java`

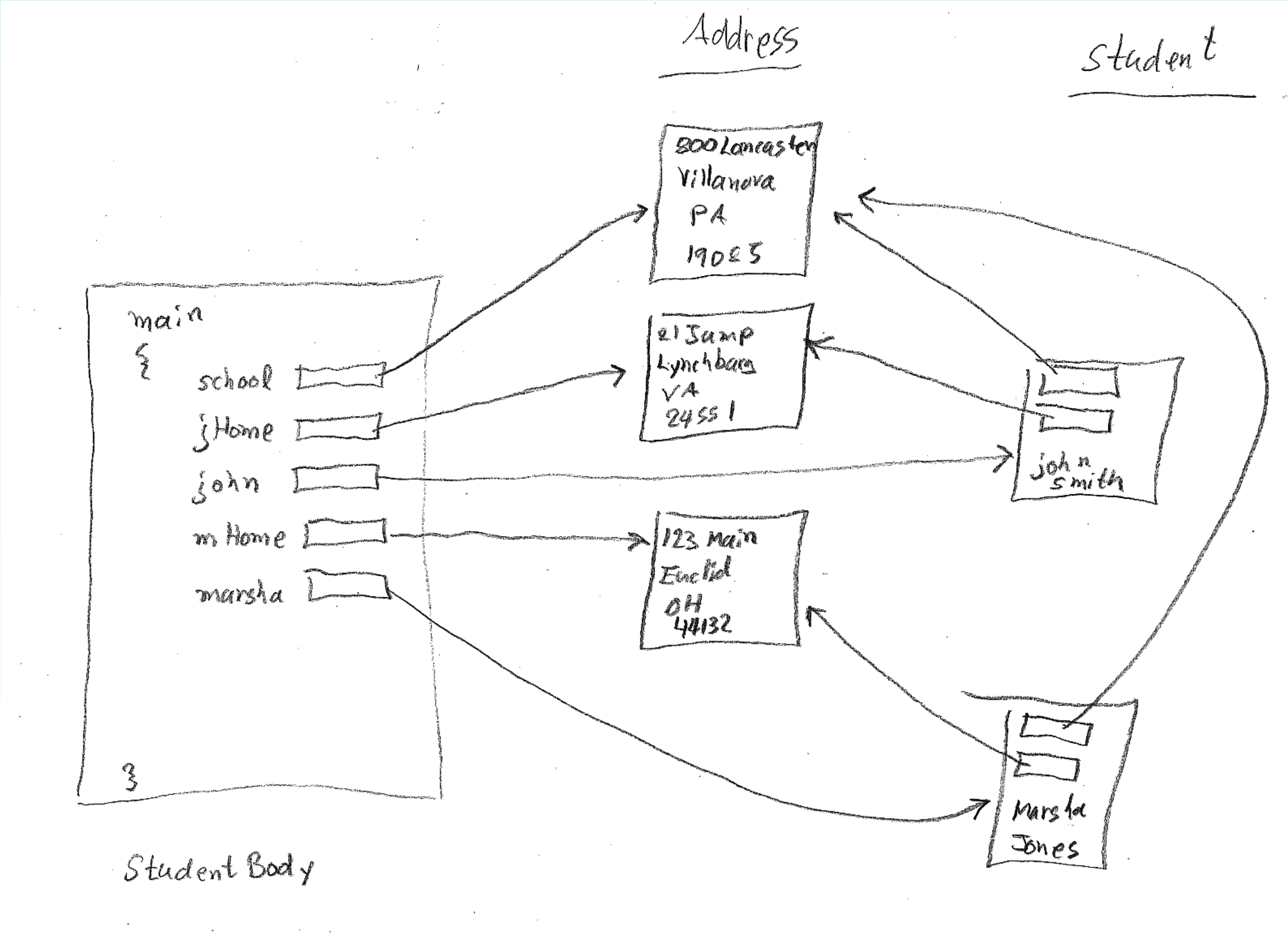
Aggregation in UML



String objects are not included in this UML diagram.
Pretend they are in the object.

UML shows relationships between classes (not between objects).

```java
//********************************************************************
//  StudentBody.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an aggregate class.
//********************************************************************

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);   // uses toString() method of john
      System.out.println();
      System.out.println(marsha); // uses toString() method of marsha
   }
}
```


Address
Student
800 Lancaster
Villanova
PA
19085
21 Jump
Lynchburg
VA
24551
123 Main
Euclid
OH
44132
main
{
school
jHome
john
mHome
marsha
}
john smith
Marsha Jones
Student Body

```
//********************                    **********************
//   StudentBody.java
//
//   Demonstrates the
//********************                    **********************

public class StudentB
{
   //------------------                    ----------------------
   //   Creates some A                    and prints them.
   //------------------                    ----------------------
   public static void
   {
      Address school =                    er Ave.", "Villanova",
                                          ;
      Address jHome =                     et", "Lynchburg",

      Student john =                      ", jHome, school);

      Address mHome =                     eet", "Euclid", "OH",
                                 44132);
      Student marsha = new Student("Marsha", "Jones", mHome, school);

      System.out.println(john);
      System.out.println();
      System.out.println(marsha);
   }
}
```

Output

```
John Smith
Home Address:
21 Jump Street
Lynchburg, VA  24551
School Address:
800 Lancaster Ave.
Villanova, PA  19085

Marsha Jones
Home Address:
123 Main Street
Euclid, OH  44132
School Address:
800 Lancaster Ave.
Villanova, PA  19085
```

```java
//********************************************************************
//  Student.java       Author: Lewis/Loftus
//
//  Represents a college student.
//********************************************************************

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;
   }
```

Put references to Address objects
Into this object.

```java
   //-----------------------------------------------------------------
   //  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;
   }
}
```

```java
//********************************************************************
//  Address.java       Author: Lewis/Loftus
//
//  Represents a street address.
//********************************************************************

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;
   }
```

```java
   //-----------------------------------------------------------------
   //  Returns a description of this Address object.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result;

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

      return result;
   }
}
```

```
public class Address
{
   private String streetAddress, city, state;
   private long zipCode;

   public Address(String street, String town, String st, long zip)
   {
      streetAddress = street;
      city = town;
      state = st;
      zipCode = zip;
   }

   public String toString()
   {
      String result;

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

      return result;
   }
}
```

```java
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;
   }

   public String toString()
   {
      String result;

      result = firstName + " " + lastName + "\n";
      result += "Home Address:\n" + homeAddress + "\n";
      result += "School Address:\n" + schoolAddress;

      return result;
   }
}
```

```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);   // uses toString() method of john
      System.out.println();
      System.out.println(marsha); // uses toString() method of marsha
   }
}
```

The this Reference

- The `this` reference allows an object to refer to itself
- The `this` reference, used inside a method, refers to the object that contains the method
- Suppose the `this` reference is used inside a method called `setX`
- Two different objects run that method:

```
obj1.setX(3);
obj2.setX(23);
```

- In the first invocation, the `this` reference inside setX refers to `obj1`; in the second it refers to `obj2`

The this Reference

```
class Thing
{
    int x;
    . . .
    void setX( int x)
    {
        this.x = x;
    }
    . . .
}
```

```
class Tester
{
   public static void main
              ( String[] args )
   {
       obj1 = new Thing();
       obj2 = new Thing();

       obj1.setX( 3 );
       obj2.setX( 23 );
   }

}
```

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 `Address` could have been written:

```
public Address(String streetAddress, String city,
               String state, long zipCode )
   {
      this.streetAddress = streetAddress ;
      this.city    = city ;
      this.state   = state ;
      this.zipCode = zipCode ;
   }
```

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Parameters

Overloading

§7.5 Interfaces

- A Java *interface* is a collection of abstract methods and constants
  - it is a description of what will be in a class (a description of a description: "meta")
  - a set of requirements that should be in a blueprint
  - like saying a blueprint for a house must include kitchen, bathroom, bed room, living room
    - but the details of implementation are not specified
    - The blueprint itself does that
- An *abstract method* is a method header without a method body

Interfaces

- An abstract method is declared using the modifier `abstract`, but because all methods in an interface are abstract automatically, usually it is left out.
- An interface lists a set of methods that a class must implement.

Interfaces

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

`interface` is a reserved word

```
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

A class (say CanDo) that implements **`Doable`** will include doThis(), doThat(), doThis2(), and doTheOther() and possibly other methods.

So objects of type CanDo will have these methods and possibly other methods.

Several classes can implement **`Doable`**.

Interfaces

- An interface cannot be directly instantiated
  - But the classes that implement it can be instantiated
- Methods in an interface have public visibility by default
- 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 implements an interface, it must define all methods in the interface

Interfaces

`implements` is a reserved word

```
public class CanDo implements Doable
{
   public void doThis()
   {
      // whatever
   }

   public void doThat()
   {
      // whatever
   }

   // etc.
}
```

Each method listed in `Doable` is given a definition

Interfaces

- In addition to abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants
- A class that implements an interface can implement methods not in the interface, as well
- See `Complexity.java`
- See `Question.java`
- See `MiniQuiz.java`

```java
//********************************************************************
//  Complexity.java       Author: Lewis/Loftus
//
//  Represents the interface for an object that can be assigned an
//  explicit complexity.
//********************************************************************

public interface Complexity
{
   public void setComplexity(int complexity);
   public int  getComplexity();
}
```

```java
//********************************************************************
//  Question.java       Author: Lewis/Loftus
//
//  Represents a question (and its answer).
//********************************************************************

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;
   }
```

```java
//-----------------------------------------------------------------
//  Sets the complexity level for this question.
//  --- as required by the Complexity interface
//-----------------------------------------------------------------
public void setComplexity(int level)
{
   complexityLevel = level;
}

//-----------------------------------------------------------------
//  Returns the complexity level for this question.
//  --- as required by the Complexity interface
//-----------------------------------------------------------------
public int getComplexity()
{
   return complexityLevel;
}

//-----------------------------------------------------------------
//  Returns the question.
//-----------------------------------------------------------------
public String getQuestion()
{
   return question;
}
```

```java
   //-----------------------------------------------------------------
   //  Returns the answer to this question.
   //-----------------------------------------------------------------
   public String getAnswer()
   {
      return answer;
   }

   //-----------------------------------------------------------------
   //  Returns true if the candidate answer matches the answer.
   //-----------------------------------------------------------------
   public boolean answerCorrect(String candidateAnswer)
   {
      return answer.equals(candidateAnswer);
   }

   //-----------------------------------------------------------------
   //  Returns this question (and its answer) as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      return question + "\n" + answer;
   }
}
```

```java
//********************************************************************
//  MiniQuiz.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a class that implements an interface.
//********************************************************************

import java.util.Scanner;

public class MiniQuiz
{
   //-----------------------------------------------------------------
   //  Presents a short quiz.
   //-----------------------------------------------------------------
   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);
```

```
      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());
   }
}
```

```java
      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
```

Implementing Interfaces

- 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

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

API Interfaces

- The Java API contains many helpful interfaces
- The `Comparable` interface contains one abstract method called `compareTo`, which compares two objects
- The `String` class implements `Comparable`, which uses lexicographic order
  - We discussed the `compareTo` method of the `String` class in Chapter 5

The Comparable Interface

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

```
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 that `obj2`,
  - 0 if they are equal,
  - and positive if `obj1` is greater than `obj2`
- It's up to the programmer of a class that implements `Comparable` to determine what makes one object less than another

The Iterator Interface

- As in Chapter 5, an iterator is an object that can process a collection of objects one at a time
- Scanner implements the `Iterator` interface
- An iterator implements 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

The Iterable Interface

- Another interface, `Iterable`, says that an object provides an iterator
- The `Iterable` interface 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:
  – an `Iterator` object performs an iteration
  – an `Iterable` object provides an iterator on request

Why 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

Outline

Software Development Activities

Identifying Classes and Objects

Static Variables and Methods

Class Relationships

Interfaces

Jump to slide 123

Parameters

Overloading

☺ Enumerated Types Revisited

☺ Method Design

☺ §7.6 Enumerated Types

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

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

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

```
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`

☺

```
//********************************************************************
//  Season.java       Author: Lewis/Loftus
//
//  Enumerates the values for Season.
//********************************************************************

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

   private String span;
```

continue

☺

continue

```
   //-----------------------------------------------------------------
   //  Constructor: Sets up each value with an associated string.
   //-----------------------------------------------------------------
   Season(String months)
   {
      span = months;
   }

   //-----------------------------------------------------------------
   //  Returns the span message for this value.
   //-----------------------------------------------------------------
   public String getSpan()
   {
      return span;
   }
}
```

☺

```java
//********************************************************************
//  SeasonTester.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a full enumerated type.
//********************************************************************

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());
   }
}
```

☺

```
//***********                                            ***************
//   SeasonTes
//
//   Demonstra
//***********                                            ***************

public class
{
   //-----------------------------------------------------------------
   //  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());
   }
}
```

Output

```
winter  December through February
spring  March through May
summer  June through August
fall    September through November
```

☺ 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

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

☺ §7.7 Method Design

- High-level design includes:
  - identifying classes and objects
  - assigning responsibilities
- After doing high-level design, you need to address low-level issues such as the design of methods
- Careful planning is needed to make sure methods 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 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 as needed

☺ Method Decomposition

- Let's look at an example of 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

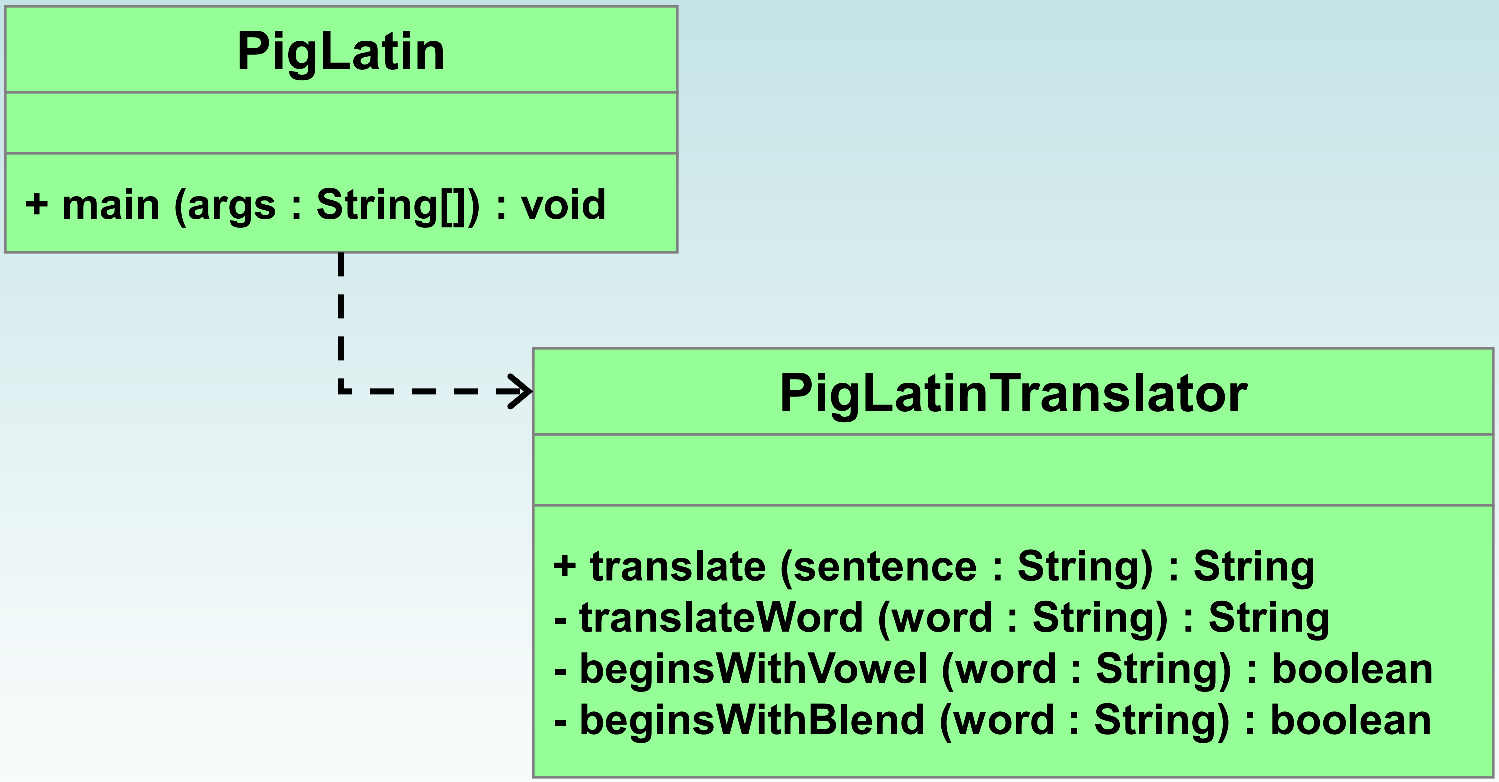
☺ Method Decomposition

- Translating a sentence is too complicated for one method to accomplish
- Look for 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 is shown by:
  - 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



```java
//********************************************************************
//  PigLatin.java       Author: Lewis/Loftus
//
//  Demonstrates the concept of method decomposition.
//********************************************************************

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 = "y";

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

```
      while (another.equalsIgnoreCase("y"))
      {
         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"));
      {
         System.out.println ();
         System.out.println ("Ente
         sentence = scan.nextLine(

         System.out.println ();
         result = PigLatinTranslat
         System.out.println ("That
         System.out.println (resul

         System.out.println ();
         System.out.print ("Transl
         another = scan.nextLine()
      }
   }
}
```

Sample Run

```
Enter a sentence (no punctuation):
Do you speak Pig Latin

That sentence in Pig Latin is:
oday ouyay eakspay igpay atinlay

Translate another sentence (y/n)? y

Enter a sentence (no punctuation):
Play it again Sam

That sentence in Pig Latin is:
ayplay ityay againyay amsay

Translate another sentence (y/n)? n
```

```java
//********************************************************************
//  PigLatinTranslator.java       Author: Lewis/Loftus
//
//  Represents a translator from English to Pig Latin. Demonstrates
//  method decomposition.
//********************************************************************

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;
   }
```

☺

```java
//-------------------------------------------------------------------
//  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.
//-------------------------------------------------------------------
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;
}
```

☺

```java
//-----------------------------------------------------------------
//  Determines if the specified word begins with a vowel.
//-----------------------------------------------------------------
private static boolean beginsWithVowel(String word)
{
   String vowels = "aeiou";

   char letter = word.charAt(0);

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

```java
   //-----------------------------------------------------------------
   //  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") );
   }
}
```

Objects as Parameters

- Parameters for a Java method are *passed by value*
- A copy of the *actual parameter* (the value passed in) is copied into the *formal parameter* (listed in the method header)
  - When the actual parameter is a primitive, a copy of the value is put into the formal parameter
  - When the actual parameter is an object reference, a copy of the reference is put into the formal parameter, so the actual parameter and the formal parameter become aliases of each other

Passing Objects to Methods

- A method with a reference parameter that points to an object can change that object.
  - The method can change objects outside of itself
- A method with a primitive parameter can change what that parameter holds, but this does not affect anything outside of the method.

```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);

      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");
   }
}
```

Step 1 (look 5 slides ahead)

Step 2 values copied into changeValues

Step 6

Nothing outside of main() can change what is in a1.
a2 and a3 point at the same object as initially,
but the object a2 points to has been changed internally.

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

public class Num
{
   private int value;

   public Num(int initialValue)  // constructor
   {
      value = initialValue;
   }

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

   public String toString()
   {
      return value + "";
   }
}
```

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

public class ParameterModifier
{
   //-----------------------------------------------------------------
   //  Modifies the parameters. f1 is a primitive.
   //  f2 and f3 point to objects that exists already.
   //-----------------------------------------------------------------
```

Step 2: f1, f2, f3 hold copies of values from main()

```java
 public void changeValues(int f1, Num f2, Num f3)
   {
      System.out.println("Before changing the values:");
```

Step 2

```java
      System.out.println("f1\tf2\tf3");
      System.out.println(f1 + "\t" + f2 + "\t" + f3 + "\n");
```

Step 3

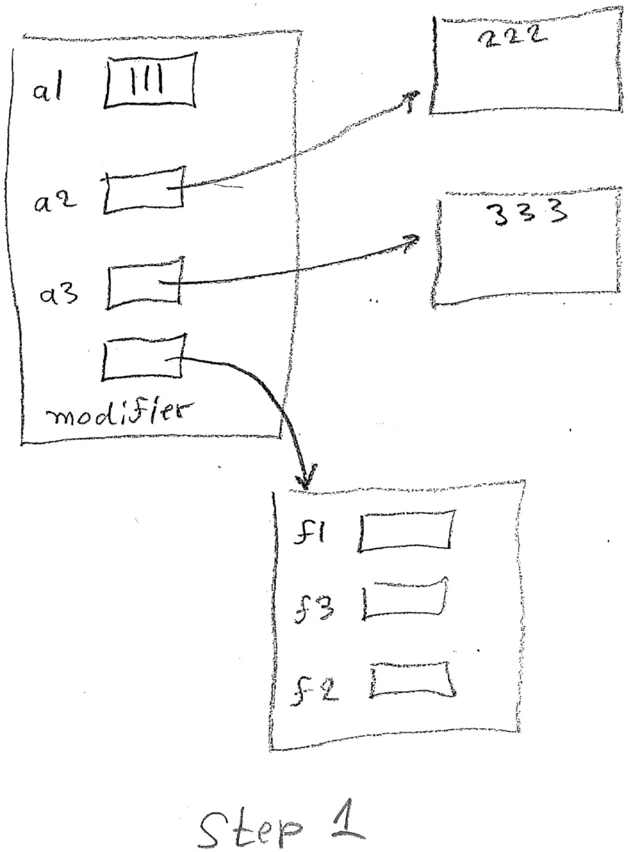
```java
      f1 = 999;
```

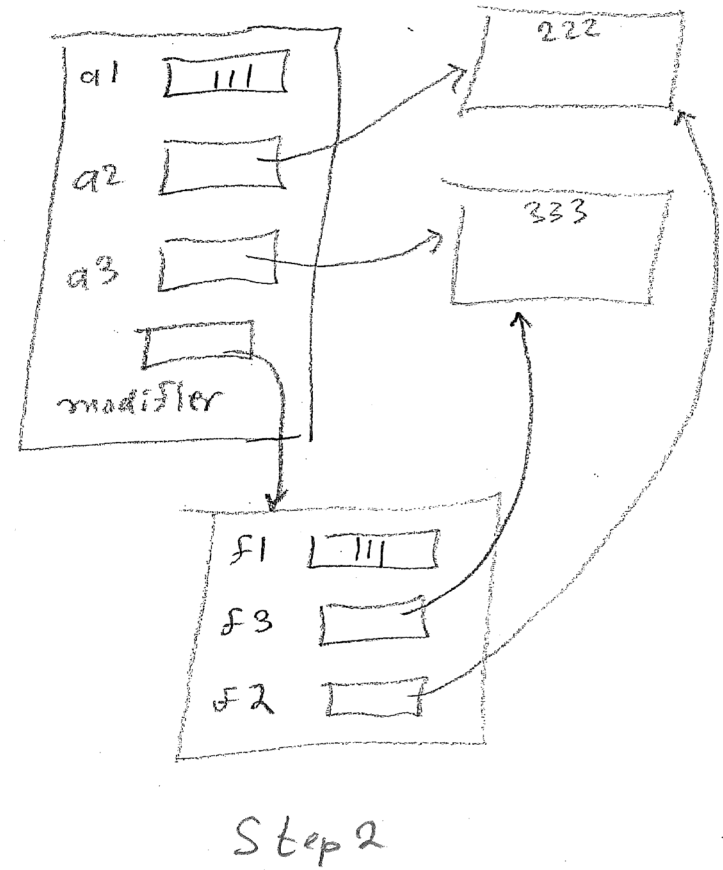
Step 4

```java
                              // change parameter (no change to main)
      f2.setValue(888);       // change the object (same one in main)
      f3 = new Num(777);      // point parameter to new object
```

Step 5

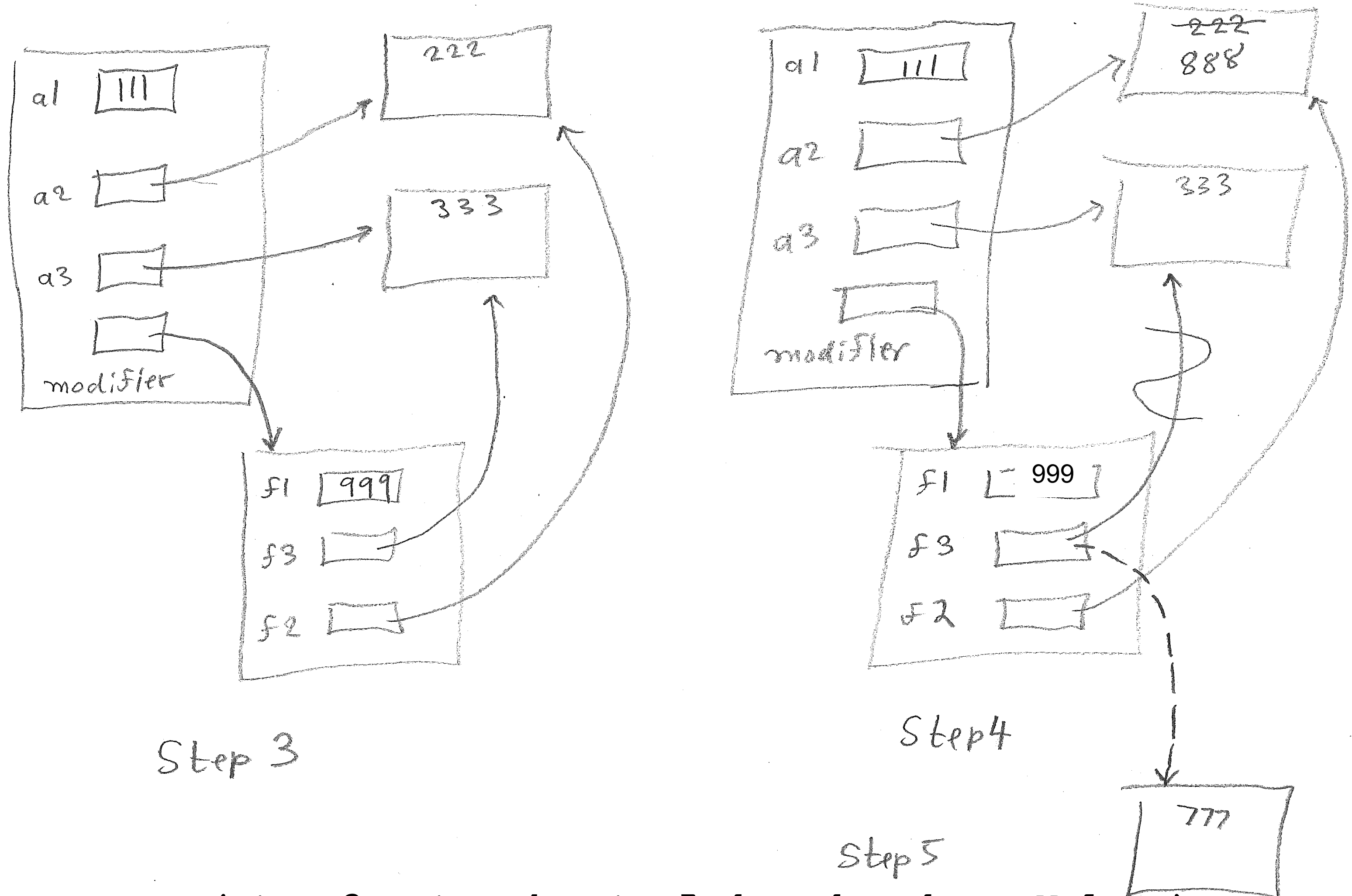
```java
                              // (object in main not affected)

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






```
>>Before calling changeValues:<<
a1      a2      a3
111     222     333
```

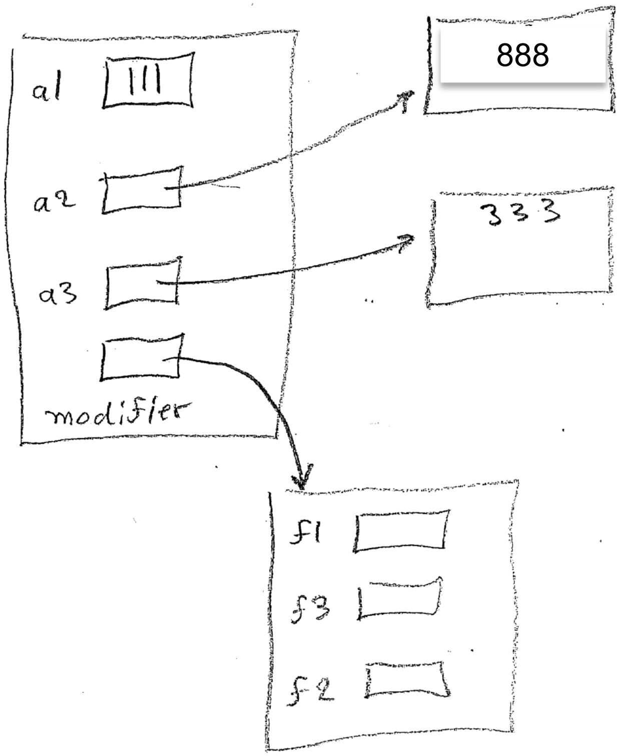
```
(change values is running)
Before changing the values:
f1      f2      f3
111     222     333
```



(step 3, step 4, step5 done by changeValues)

After changing the values:

f1	**f2**	**f3**
999	**888**	**777**



Step 6

```
Now running in main:
>>After calling changeValues:<<
a1      a2      a3
111     888     333
```

Output

Step 1

```
>>Before calling changeValues:<<
a1      a2      a3
111     222     333
```

Step 2

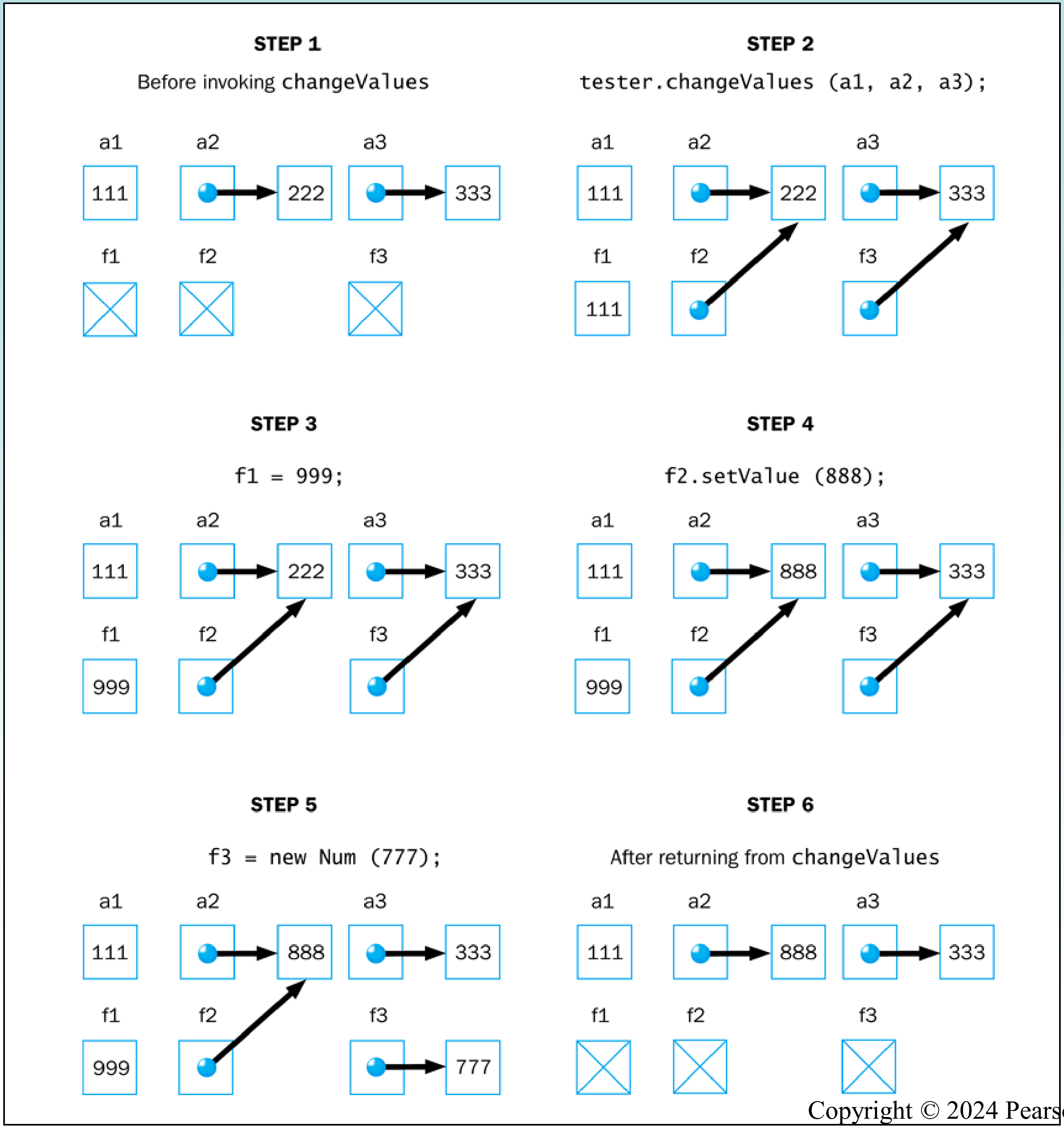
```
Before changing the values:
f1      f2      f3
111     222     333
```

Step 5

```
After changing the values:
f1      f2      f3
999     888     777
```

Step 6

```
>>After calling changeValues:<<
a1      a2      a3
111     888     333
```


STEP 1
Before invoking changeValues
a1
a2
a3
111
222
333
f1
f2
f3
STEP 2
tester.changeValues (a1, a2, a3);
a1
a2
a3
111
222
333
f1
f2
f3
111
STEP 3
f1 = 999;
a1
a2
a3
111
222
333
f1
f2
f3
999
STEP 4
f2.setValue (888);
a1
a2
a3
111
888
333
f1
f2
f3
999
STEP 5
f3 = new Num (777);
a1
a2
a3
111
888
333
f1
f2
f3
999
777
STEP 6
After returning from changeValues
a1
a2
a3
111
888
333
f1
f2
f3

§7.8 Method Overloading

- *Method overloading* is giving a single name to multiple definitions of a method in a class
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique (sometimes called *method header*)
- The signature includes the number, type, and order of the parameters

Method Overloading

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

```
class Eg
{
   float tryMe(int x)
   {
      return x + .375;
   }
   float tryMe(int x, float y)
   {
       return x*y;
   }
     . . .
}
```

Invocation

```
result = tryMe(25, 4.32)
```

```
class Tester
{
  public static void main (String[] args)
  {
    float x, y;
    Eg obj = new Eg();
    x = obj.TryMe( 2 ); // x gets 2.375

    y = obj.tryMe(25, 4.32) // y gets
                            // 108.0

  }
```

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 <u>not</u> 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

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

☺ §7.9 Testing

- Testing can mean many different things
- It certainly includes running a completed program with various inputs
- Some evaluations should occur before coding even begins
- The earlier we find a 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 undiscovered errors

☺ Reviews

- A *review* is a meeting in which several people examine a design document or section of code
- Often a member of a programming team presents their design to the other members
- 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 using 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. (combinatorial explosion)
- 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 would produce similar results
- Therefore a good test suite will cover all equivalence categories and focus on the boundaries between categories

☺ 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

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

☺ §7.10 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

☺ 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 <u>is</u> the program

☺ 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

☺ 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

☺ 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:

Flow Layout
Border Layout
Card Layout
Grid Layout
GridBag Layout

Defined in the AWT

Box Layout
Overlay Layout

Defined in Swing

☺ 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:

```
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

- See `LayoutDemo.java`
- See `IntroPanel.java`

```
//********************************************************************
//  LayoutDemo.java       Authors: Lewis/Loftus
//
//  Demonstrates the use of flow, border, grid, and box layouts.
//********************************************************************

import javax.swing.*;

public class LayoutDemo
{
   //-----------------------------------------------------------------
   //  Sets up a frame containing a tabbed pane. The panel on each
   //  tab demonstrates a different layout manager.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      JFrame frame = new JFrame("Layout Manager Demo");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

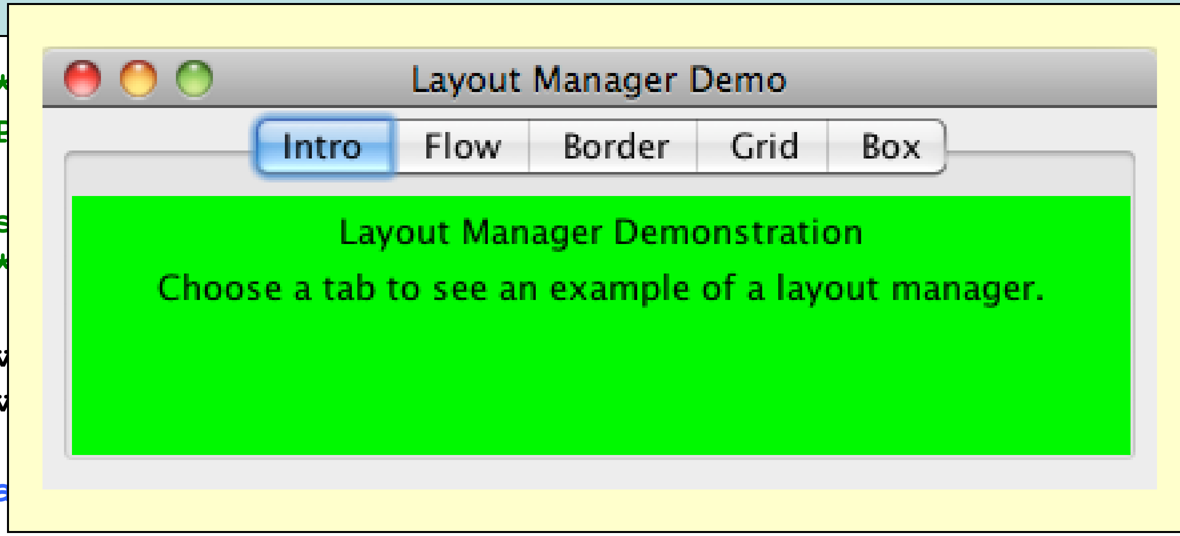
continue

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);
   }
}
```

```java
//********************************************************************
//  IntroPanel.java       Authors: Lewis/Loftus
//
//  Represents the introduction panel for the LayoutDemo program.
//********************************************************************

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);
   }
}
```

```
//*******                                          *******
//   IntroP
//
//   Repres                                        m.
//*******                                          *******

import jav
import jav

public cla
{
   //-----------------------------------------------------------------
   //  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

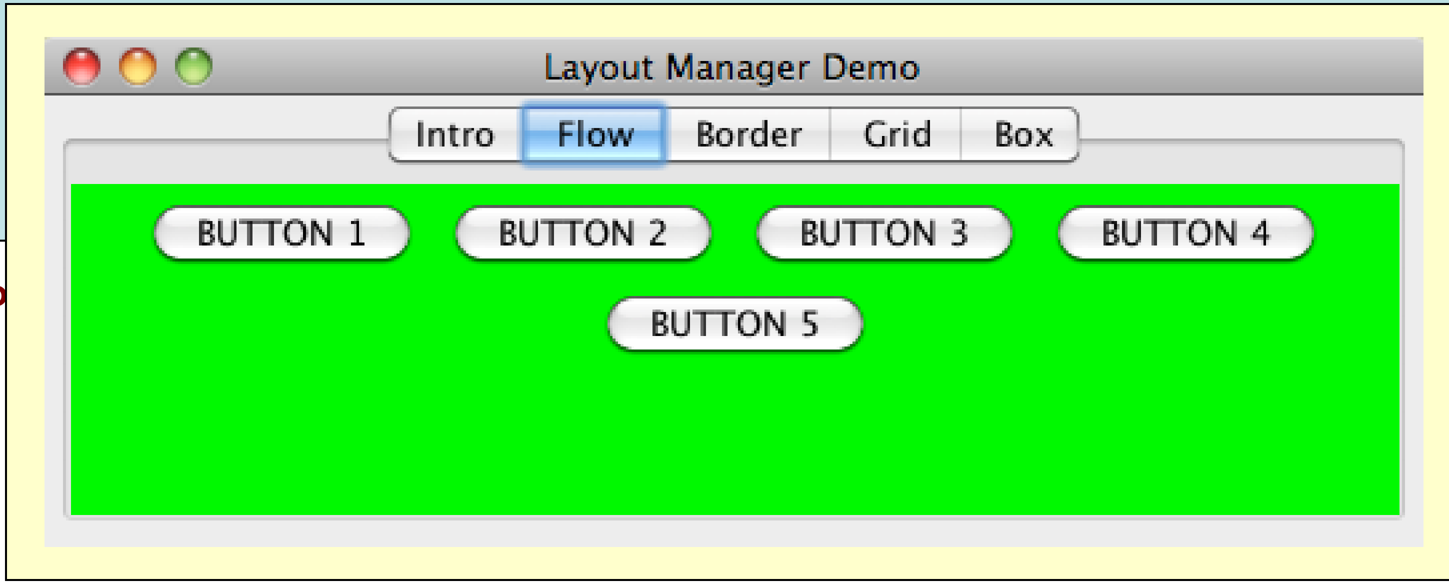
- *Flow layout* puts as many components as possible on a row, then moves to the next row
- Components are displayed in the order they are added to the container
- Each row of components is centered horizontally by default, but could also be aligned left or right
- The horizontal and vertical gaps between the components can be explicitly set
- See `FlowPanel.java`

```java
//********************************************************************
//  FlowPanel.java       Authors: Lewis/Loftus
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the flow layout manager.
//********************************************************************

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);
```

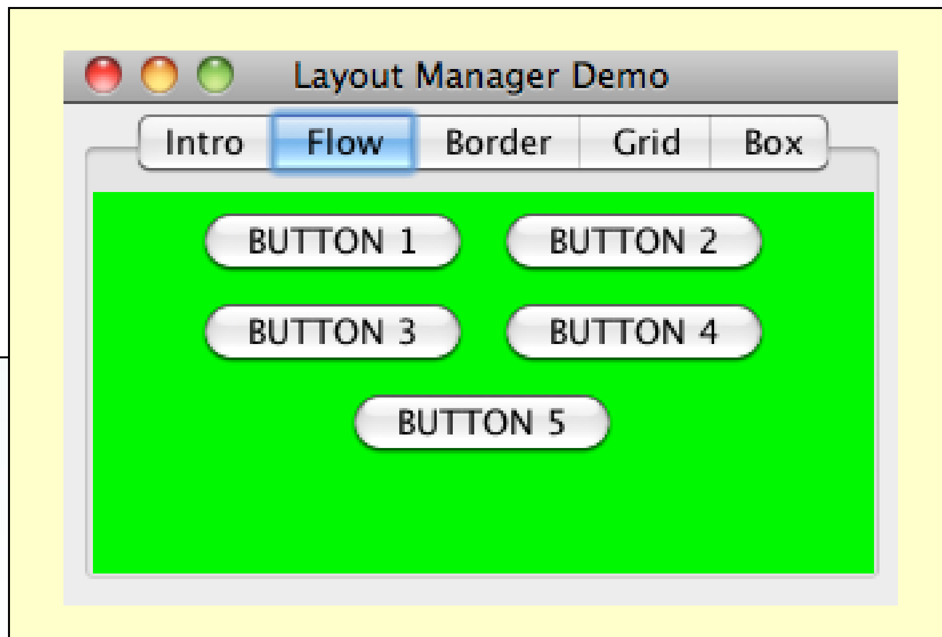
continue

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(b2);
        add(b3);
        add(b4);
        add(b5);
    }
}
```

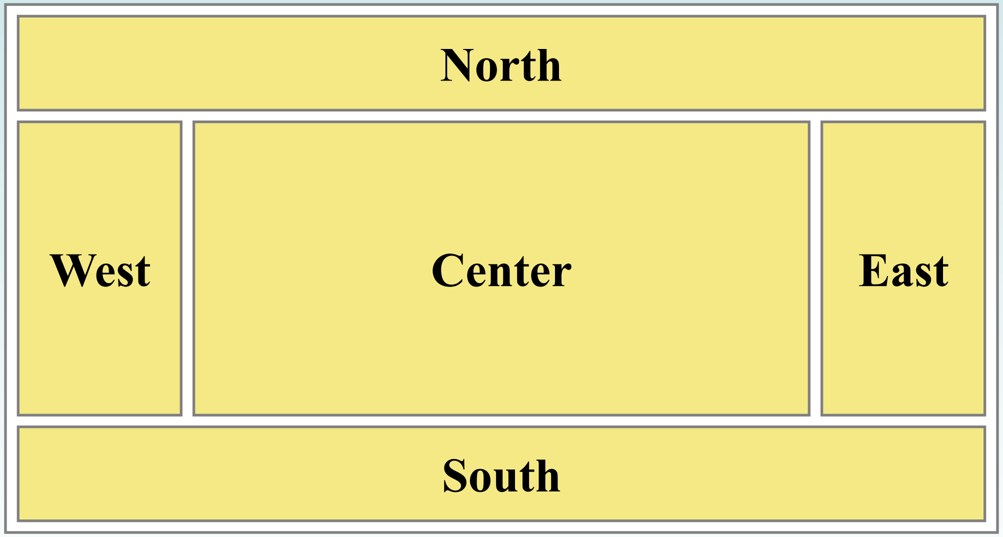



```
co

      add(b1)
      add(b2)
      add(b3)
      add(b4)
      add(b5)
   }
}
```



☺ 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`

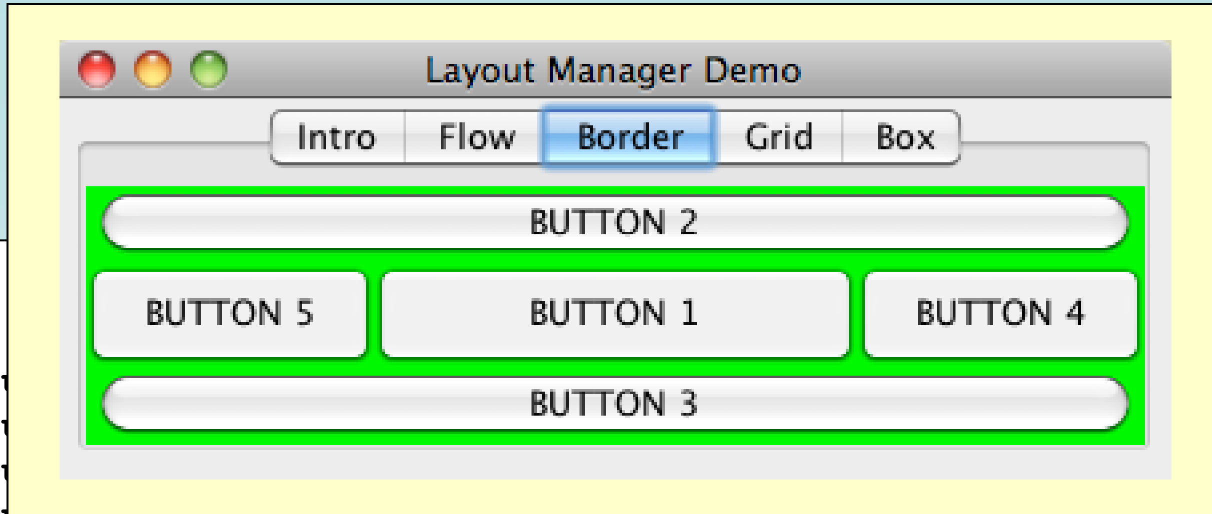
```java
//********************************************************************
//  BorderPanel.java       Authors: Lewis/Loftus
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the border layout manager.
//********************************************************************

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

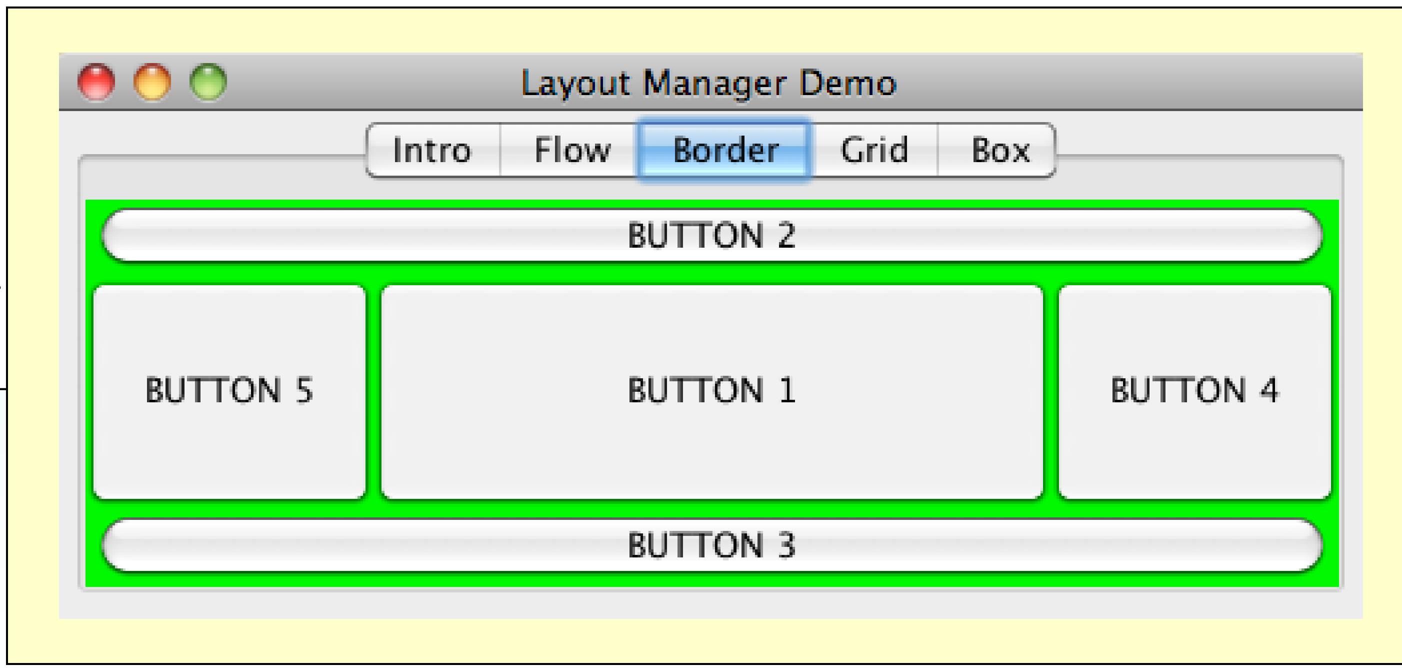
continue

```
        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);
    }
}
```

continue




```
    JB
    JB
    JB
    JButton b4 = new JButton("BUTTON 4");
    JButton b5 = new JButton("BUTTON 5");

  }
}
```



☺ 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`

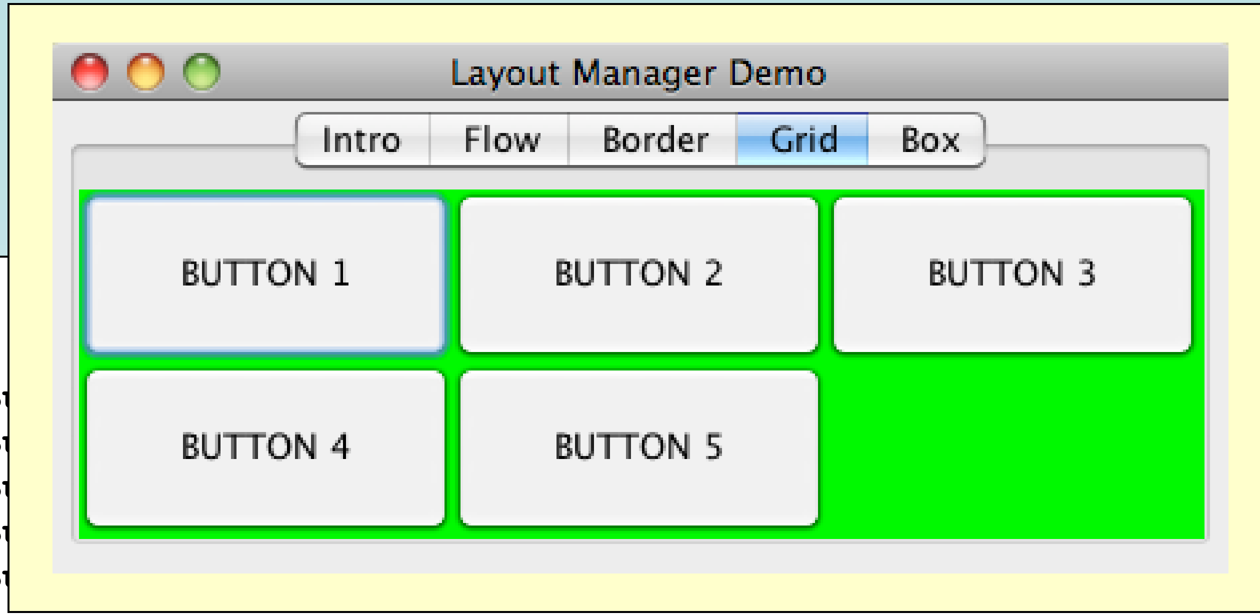
```java
//********************************************************************
//  GridPanel.java       Authors: Lewis/Loftus
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the grid layout manager.
//********************************************************************

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);
```

continue

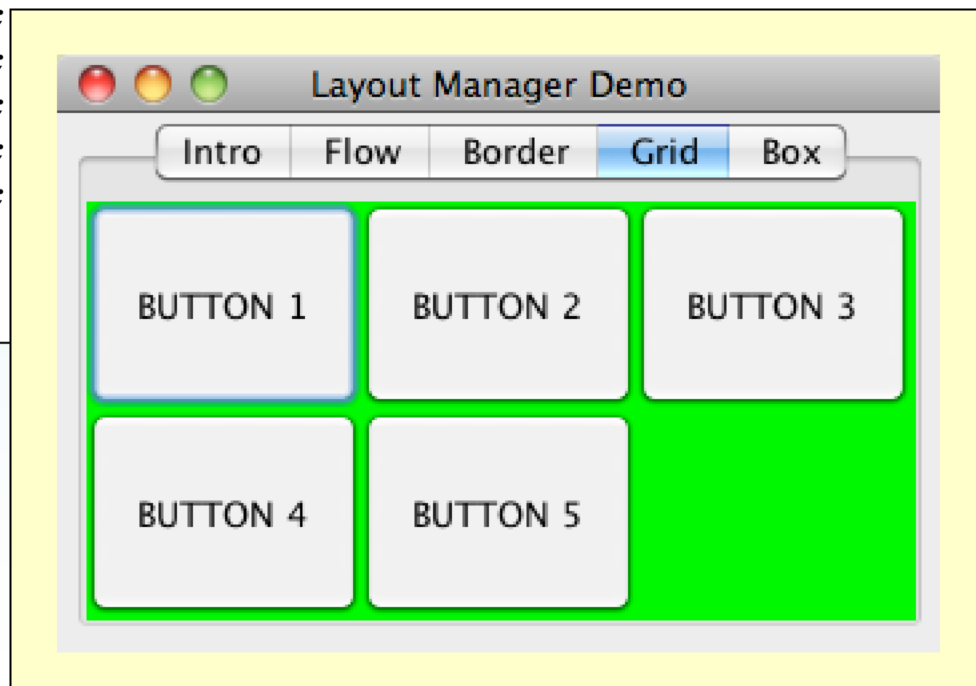
continue

```
        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(b2);
        add(b3);
        add(b4);
        add(b5);
    }
}
```

continue




```
        JBt
        JBt
        JBt
        JBt
        JBt

        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:

```
createRigidArea(Dimension d)

createHorizontalGlue()

createVerticalGlue()
```

- See `BoxPanel.java`

```java
//********************************************************************
//  BoxPanel.java       Authors: Lewis/Loftus
//
//  Represents the panel in the LayoutDemo program that demonstrates
//  the box layout manager.
//********************************************************************

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);
```

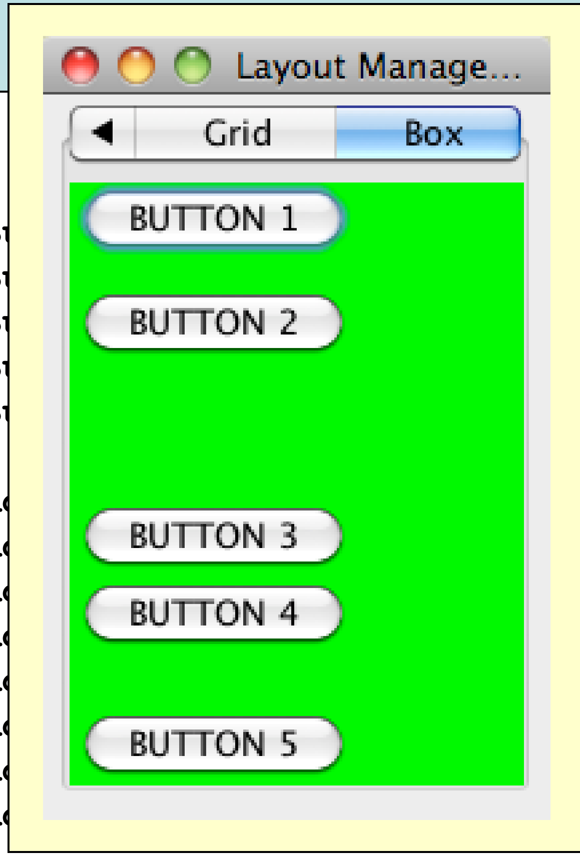
continue

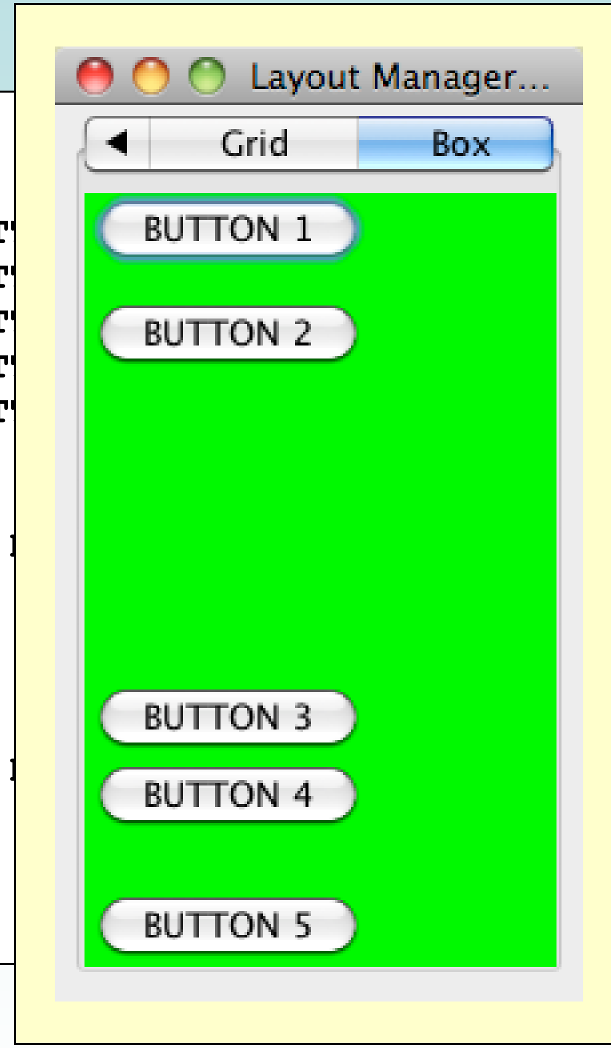
continue

```
        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

```
        JBu                                 UT
        JBu                                 UT
        JBu                                 UT
        JBu                                 UT
        JBu                                 UT

        add
        add                                 w I
        add
        add                                 ))
        add
        add
        add                                 w I
        add
    }
}
```





☺ 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`

```java
//********************************************************************
//  BorderDemo.java       Authors: Lewis/Loftus
//
//  Demonstrates the use of various types of borders.
//********************************************************************

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

continue

```
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);
```

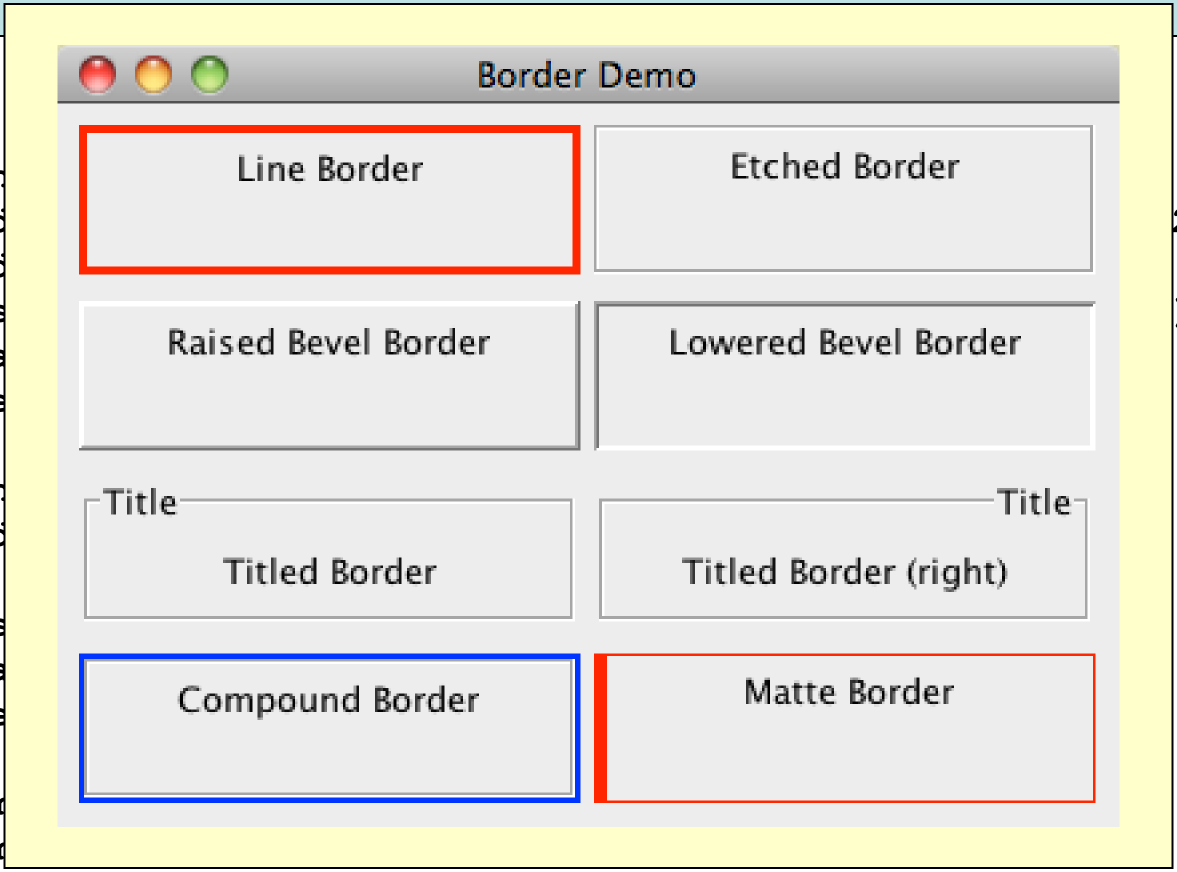
continue

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

```
        JPan
        Bord                                                  2);
        Bord
        p7.s                                                  ));
        p7.a
        pane

        JPan
        Bord

        p8.s
        p8.a
        pane

        fram
        fram
        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