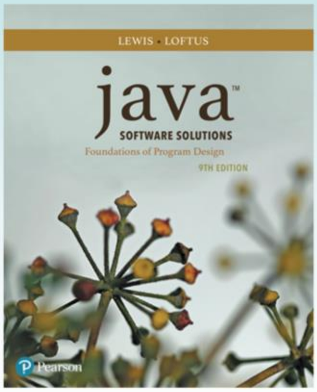
Chapter 7
Object-Oriented Design

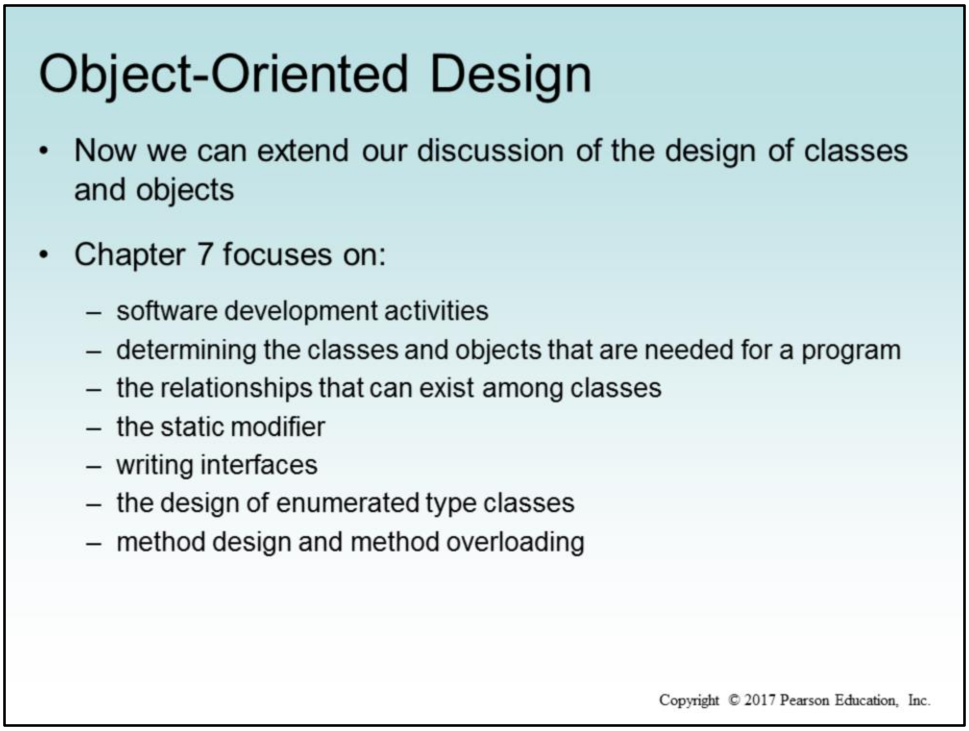


Java Software Solutions

Foundations of Program Design

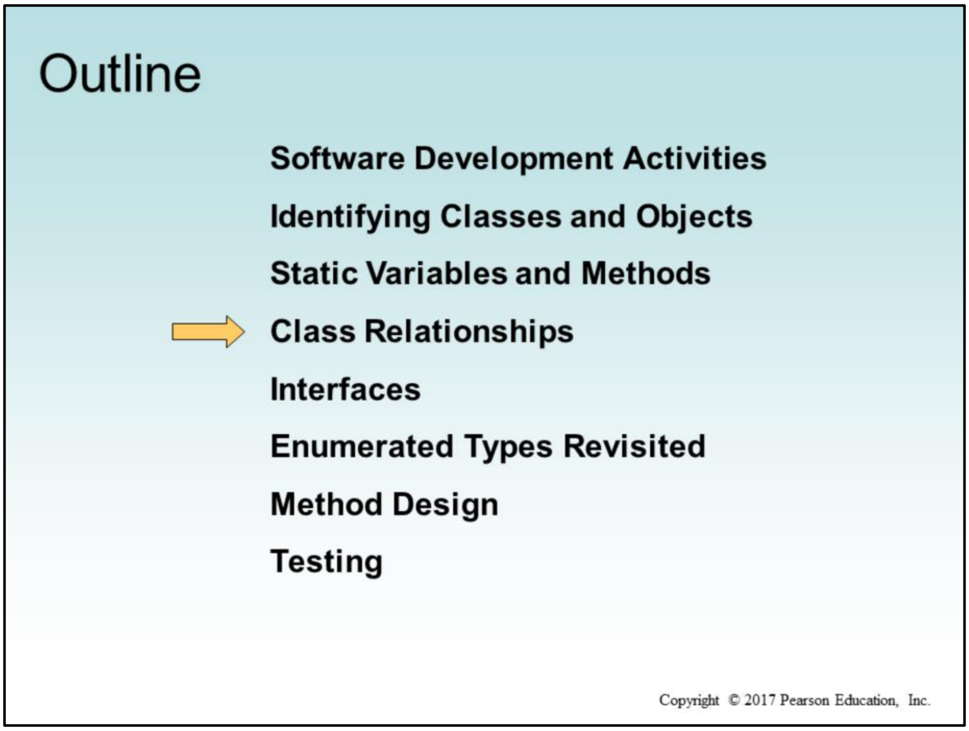
9th Edition

John Lewis
William Loftus



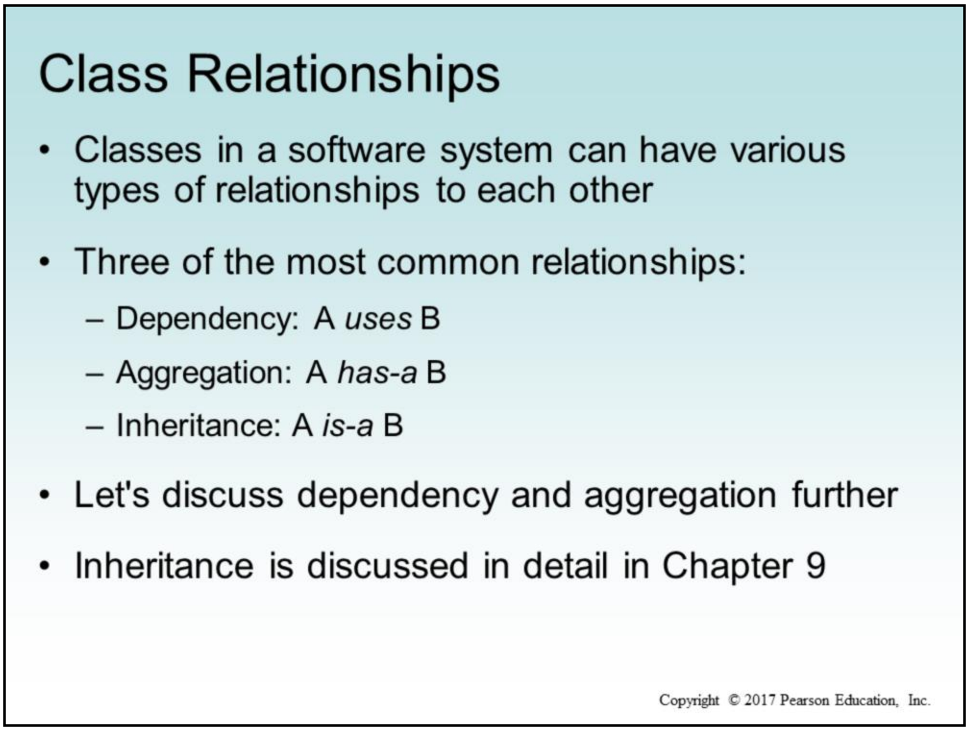
Object-Oriented Design

- 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



Outline

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

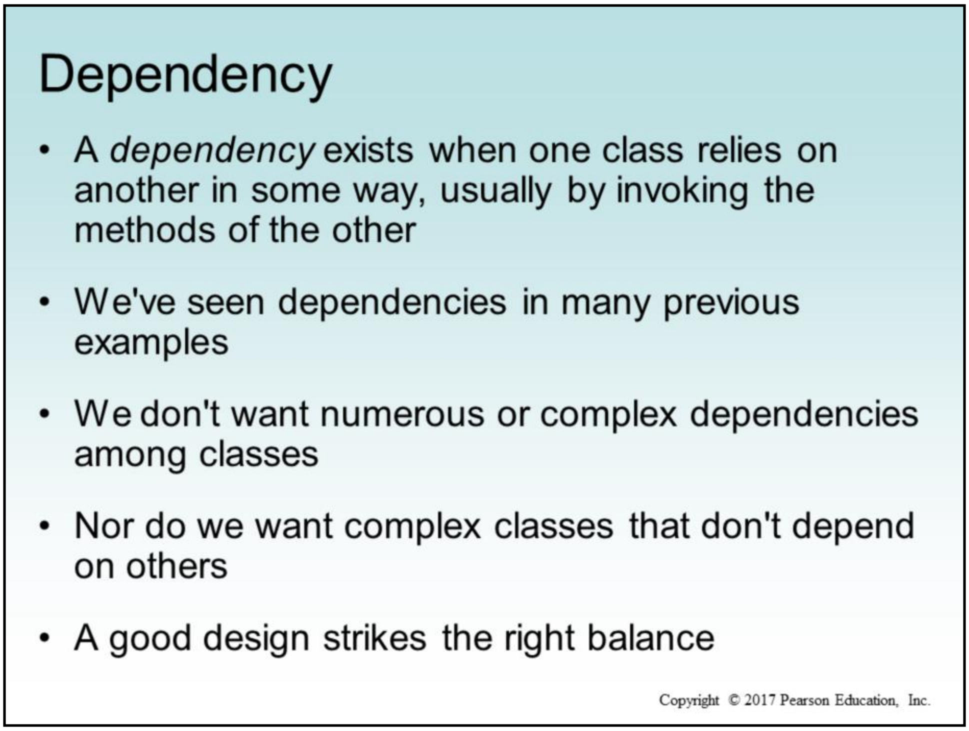


Class Relationships

- 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

-Now that as we are beginning to learn how to write our own classes, we need to consider class relationships

-Specifically we look at how objects of classes can be used and are dependent upon one another to solve problems



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

-In theory, we’ve already seen and written labs where one class uses objects from another

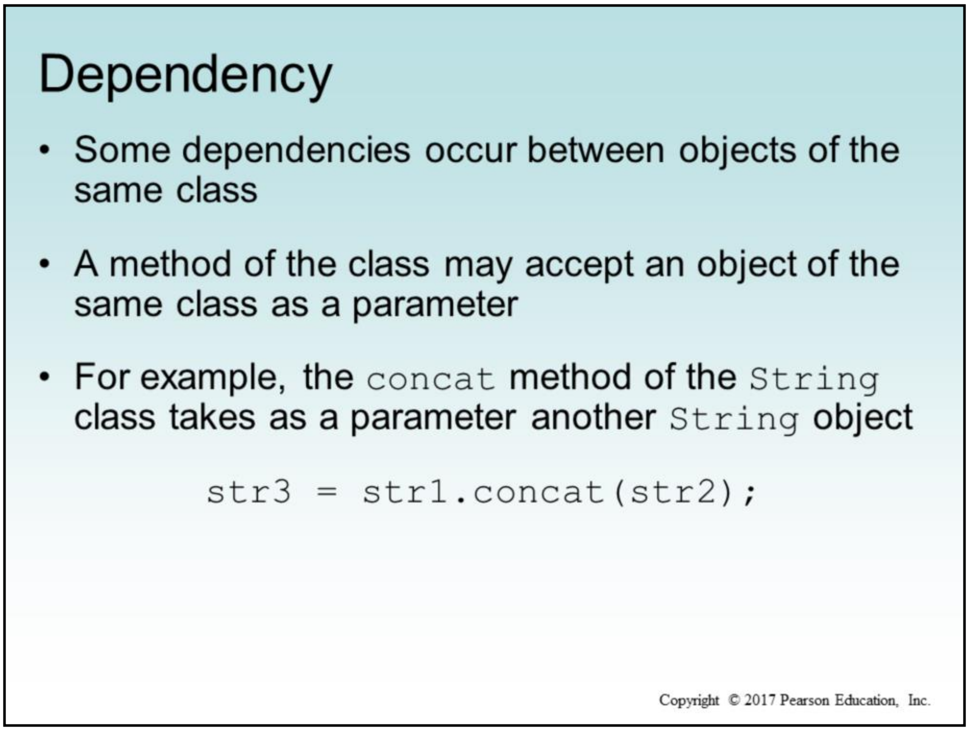
-Consider, for example, when we create and use a Scanner object in a method (e.g. main method of a driver class)

-Our main driver method is “using” another object from a different class

-As we develop more complex classes (other than driver classes), we want to consider objects we need

-We want to avoid too many inter-dependencies where classes depend on too many other different objects

-Changes to other classes of our dependent objects could force our class to change in ways we never intended



Dependency

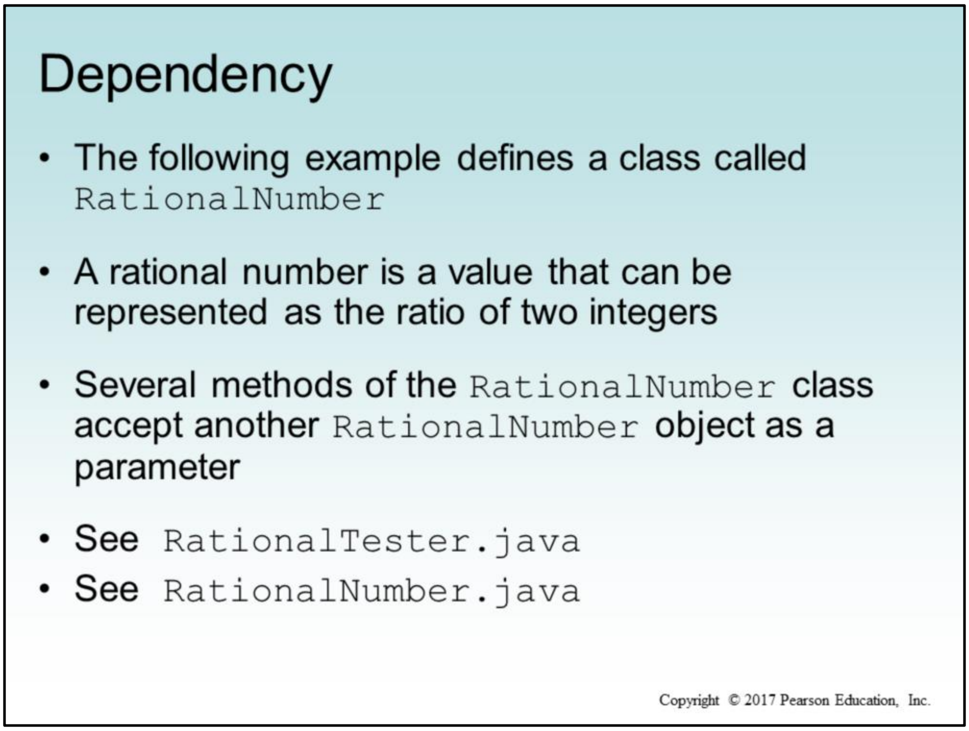
- Some dependencies occur between objects of the same class
- A method of the class may accept an object of the same class as a parameter
- For example, the `concat` method of the `String` class takes as a parameter another `String` object

```
str3 = str1.concat(str2);
```

-Some methods of our class design could actually accept objects of the class type itself as method parameters!

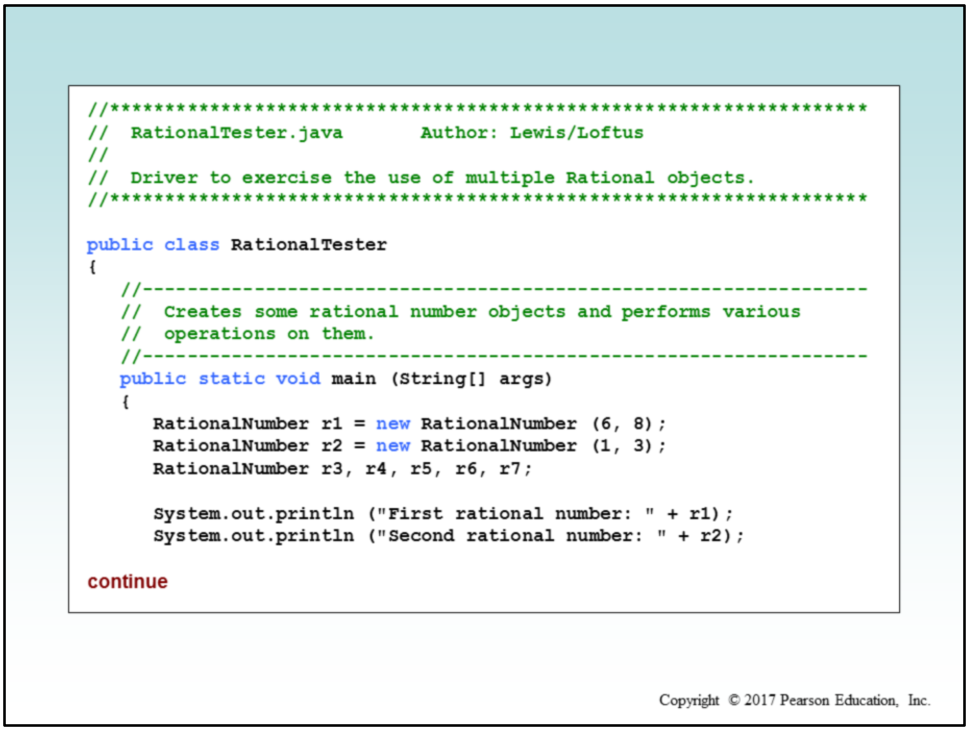
-Shown above is the concat method of the String class that takes as its parameter another String object

-This is a case where the String class depends upon itself (or uses itself) to perform tasks



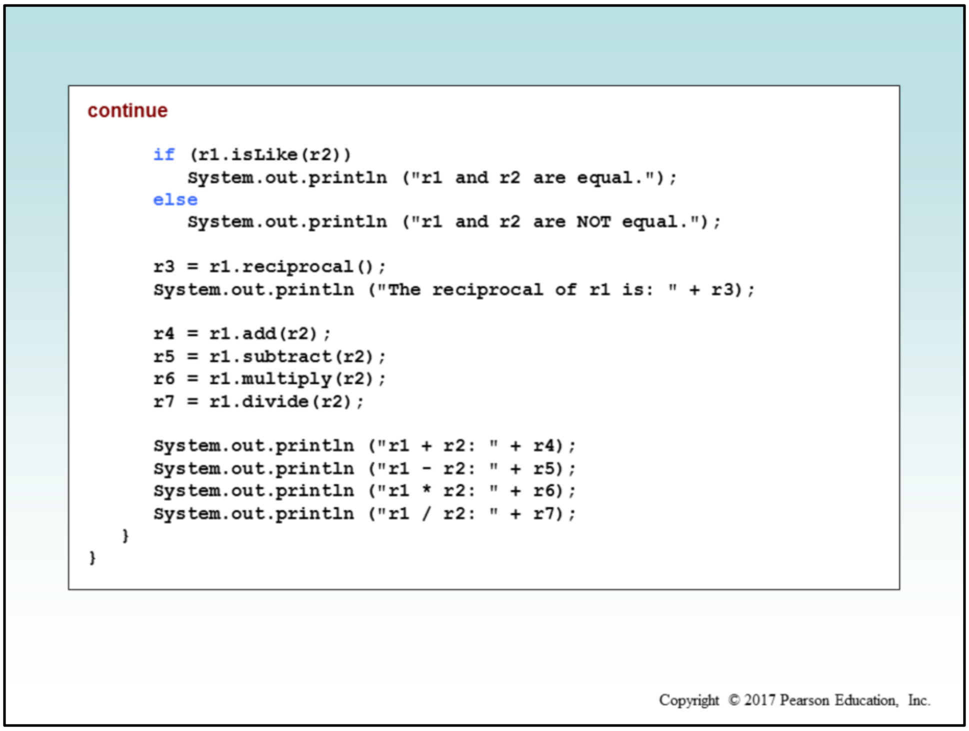
Dependency

- 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`



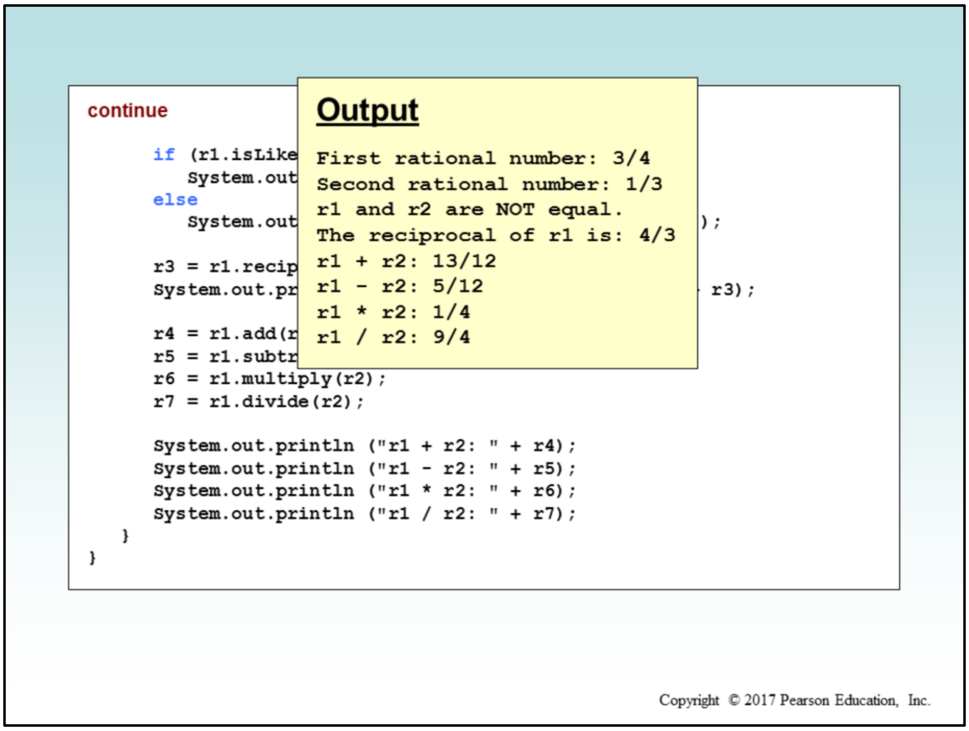
```
//********************************************************************
//  RationalTester.java       Author: Lewis/Loftus
//
//  Driver to exercise the use of multiple Rational objects.
//********************************************************************

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);
```

continue



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);
   }
}
```

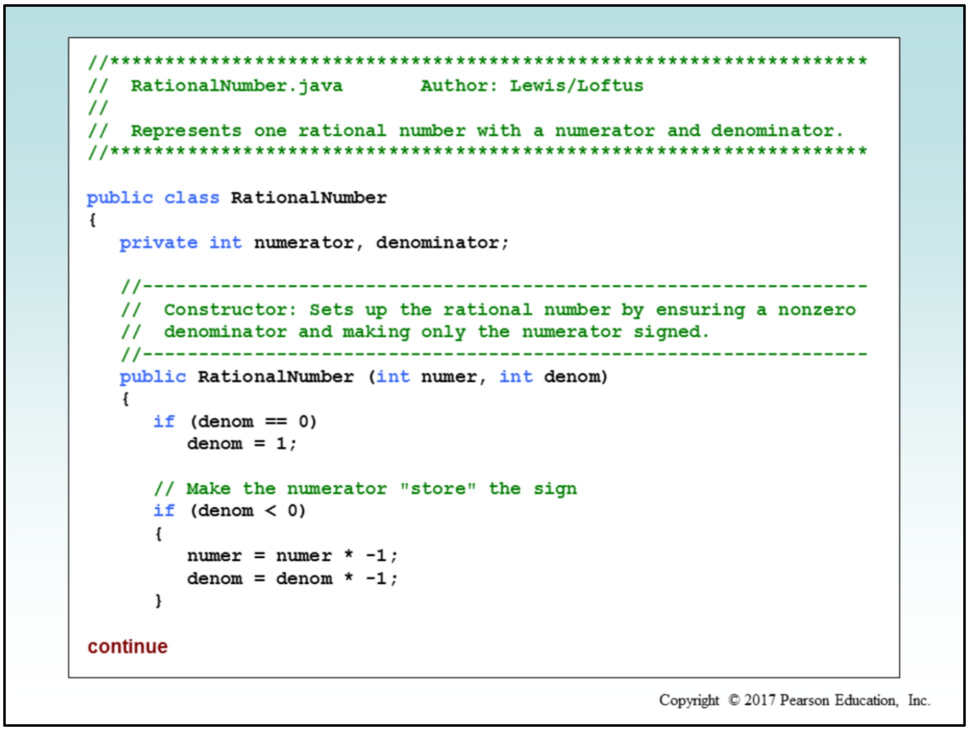


continue

```
      if (r1.isLike
         System.out
      else
         System.out                                         );

      r3 = r1.recip
      System.out.pr                                          r3);

      r4 = r1.add(r
      r5 = r1.subtr
      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);
   }
}
```

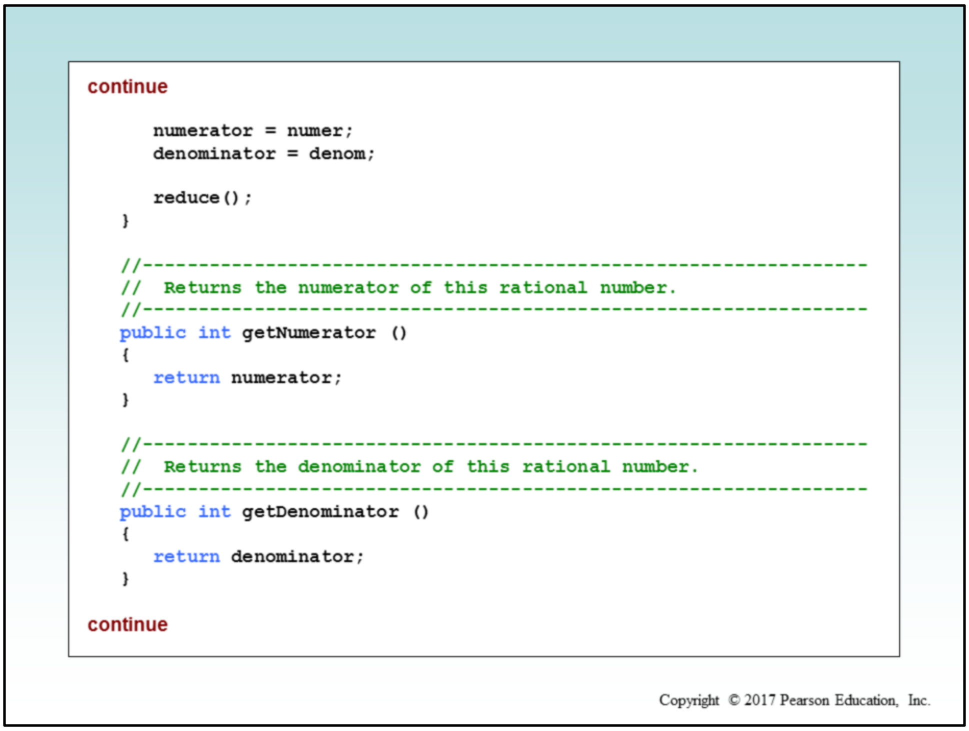
Output

```
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;
      }
```

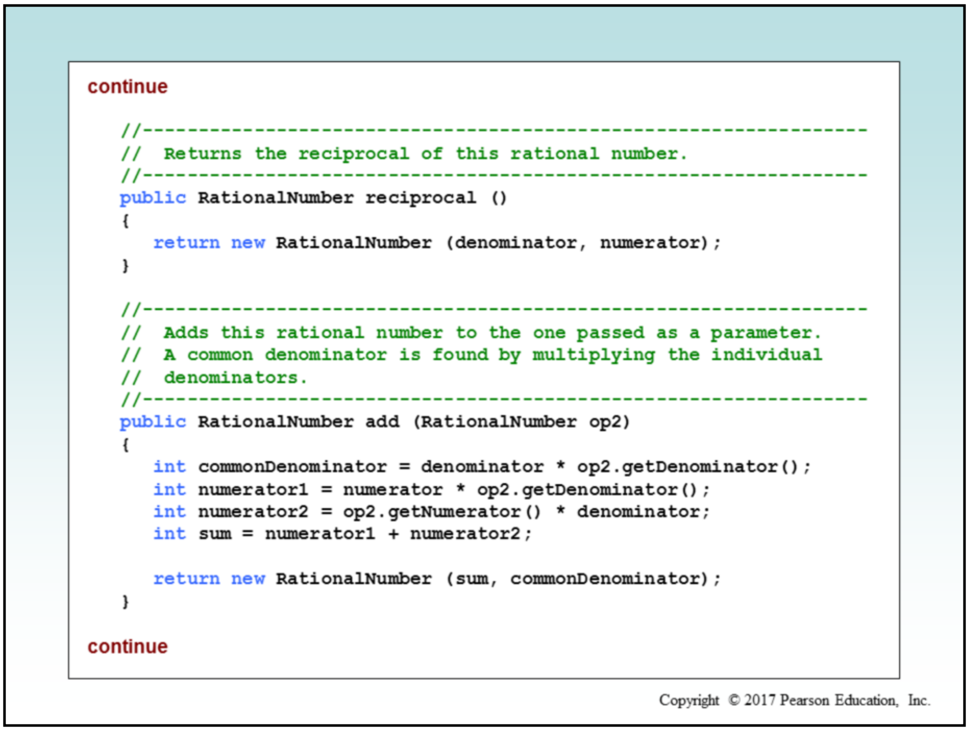
continue



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;
   }
```

continue



continue

```
   //-----------------------------------------------------------------
   //  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);
   }
```

continue

-Look closely at the add method of the RationalNumber class

-Note that it is adding **itself** to another RationalNumber (**op2**) passed as a parameter

-It is important to understand which object is which when the add method is actually called

-In the driver test program (RationalTester), we see the statement:

r4 = r1.add(r2), where r1, r2, and r4 are all RationalNumber objects

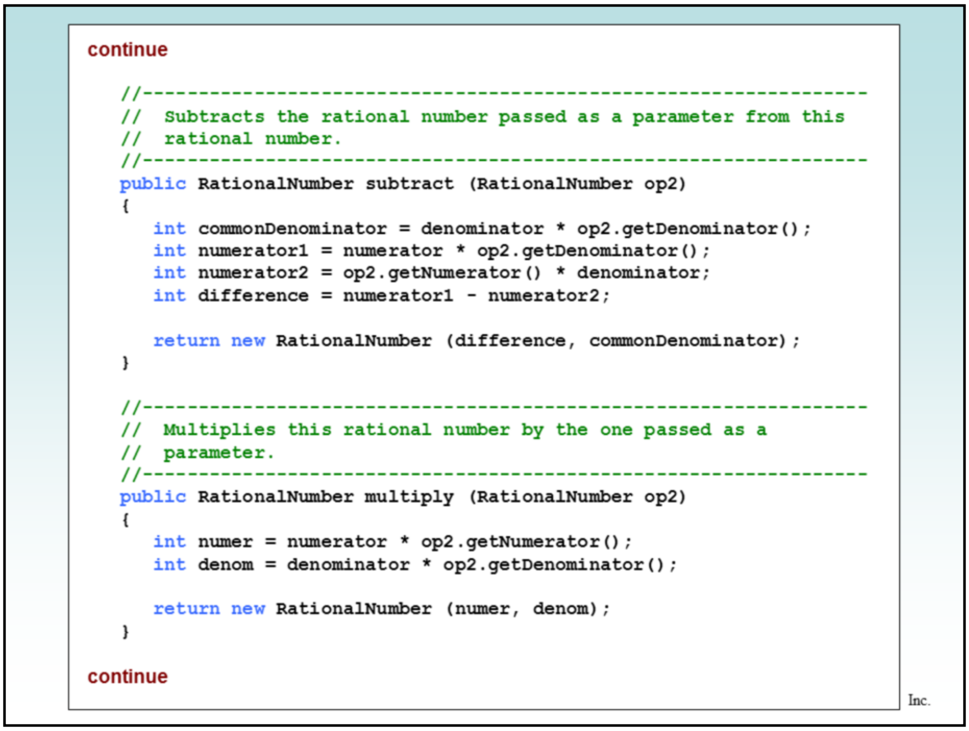
-When this add method is called, r2 becomes op2 and r1 is the object itself

-The instance variables **denominator** and **numerator** in the add method are from **r1**

-The instance variables returned from **op2.getDenominator** and **op2.getNumerator** are from **r2**

-The instance variables in a method used *without an object* are those from the object that *called* the method

-The add method computes the result in a new RationalNumber object and it is then stored in **r4**



continue

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

      return new RationalNumber (numer, denom);
   }
```

continue

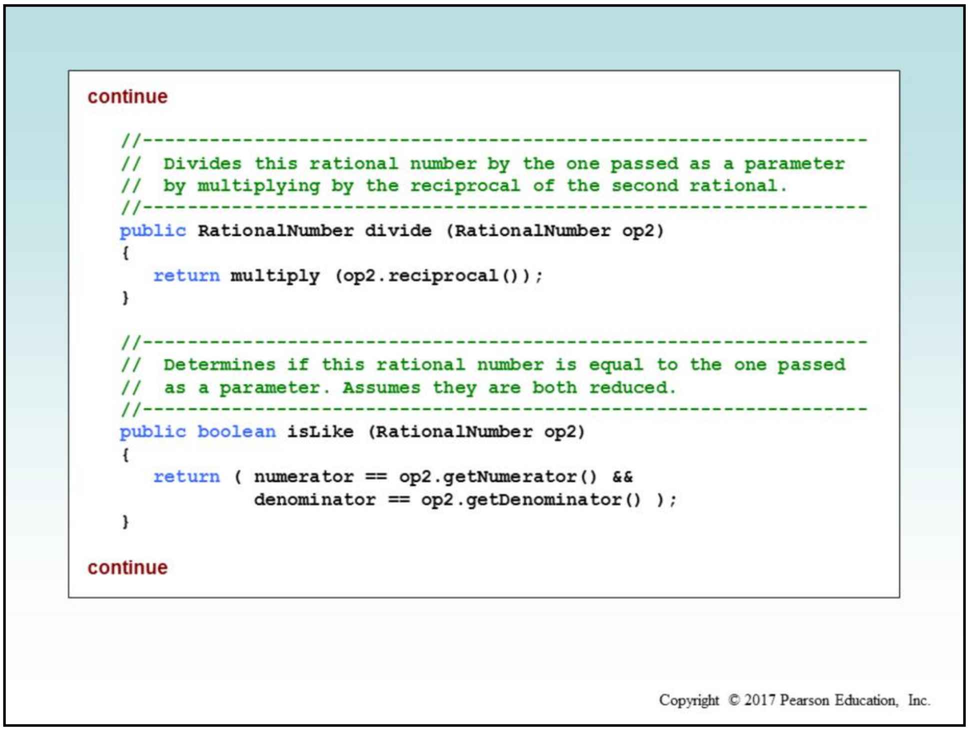
-Note the return statements in many of the methods

return new RationalNumber(….);

-This statement creates a new object and immediately returns it to the calling method

-Note you do not need to store it first in an object reference variable (although you could)

-This is sometimes referred to as returning a “nameless” object



continue

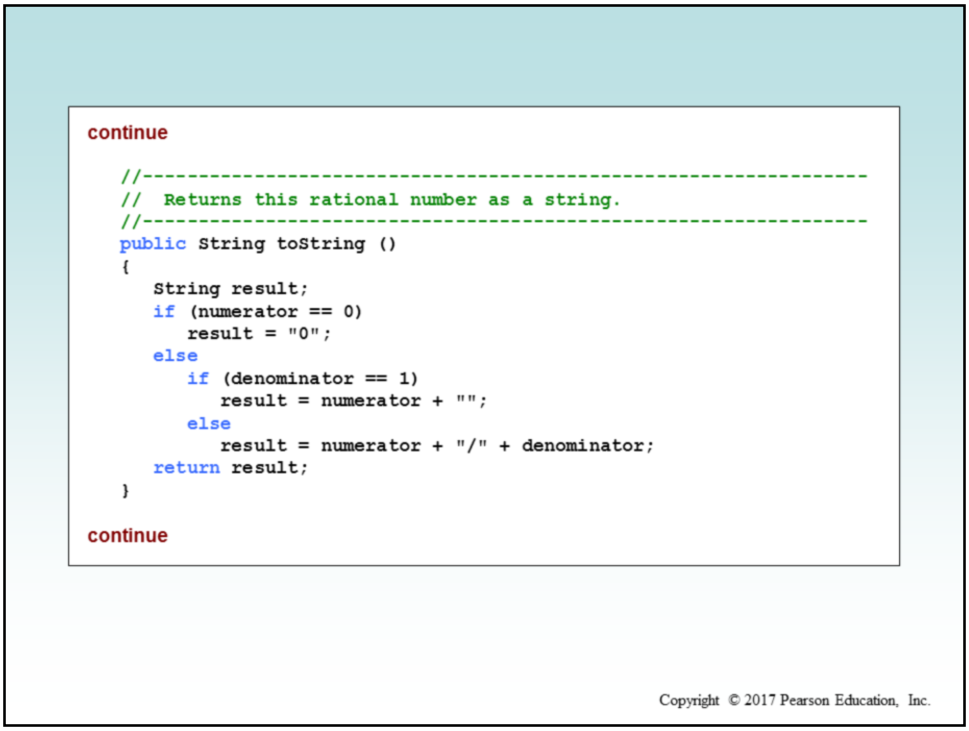
```
   //-----------------------------------------------------------------
   //  Divides this rational number by the one passed as a parameter
   //  by multiplying by the reciprocal of the second rational.
   //-----------------------------------------------------------------
   public RationalNumber 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() );
   }
```

continue

-Note the isLike method is very similar to the equals method we’ve seen in other classes

-This method determines if the object that called the method is equal to one that is passed as a parameter

-We’ll see later that we typically implement such functionality in a method named equals (instead of isLike)



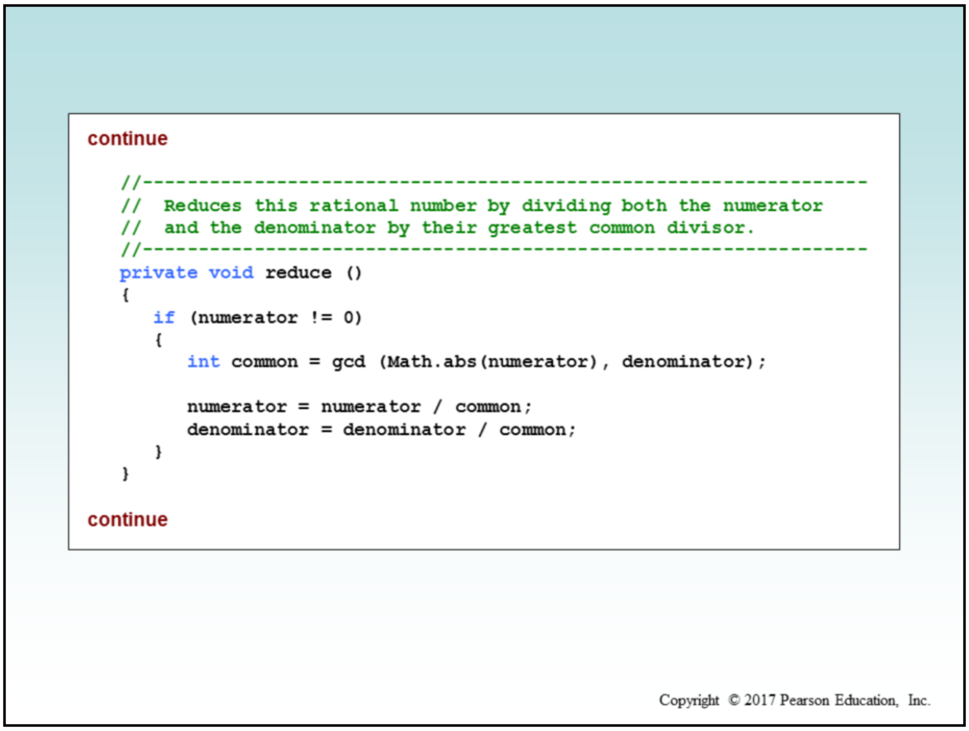
continue

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

continue

-As we’ve seen with other classes, note the implementation of the method named toString

-This method, as we’ve seen, returns a String representation of the instance variable data (what the class “knows”)



continue

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

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

continue

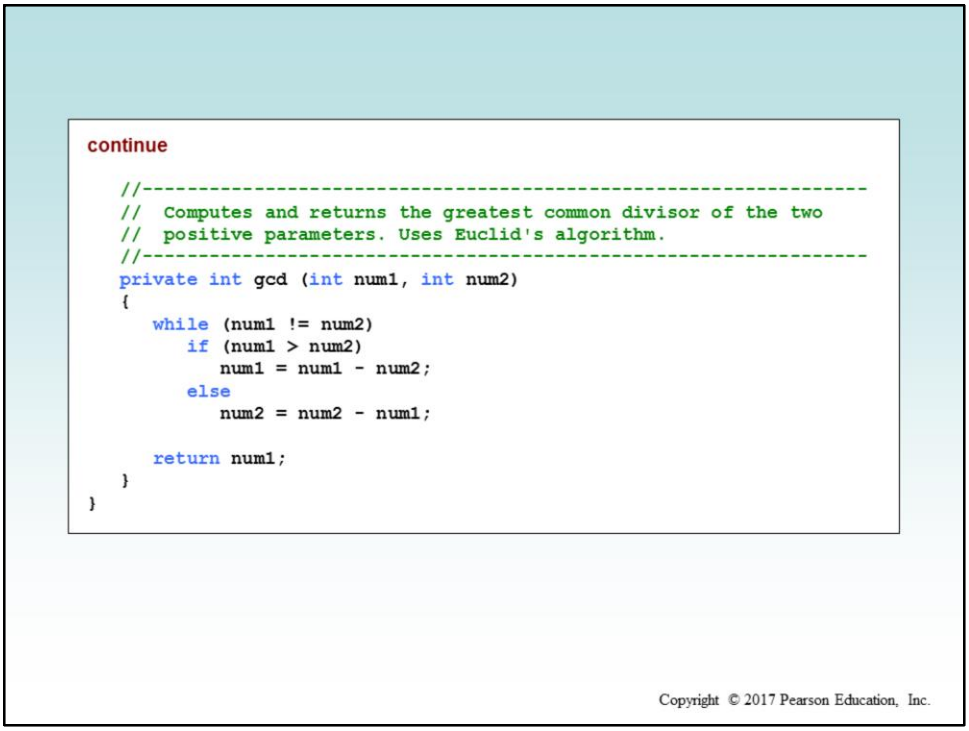
-Note that some methods are private (e.g. reduce, gcd)

-This means that methods using RationalNumber objects cannot call these methods

-Methods that are private can only be called from other methods **in the class**

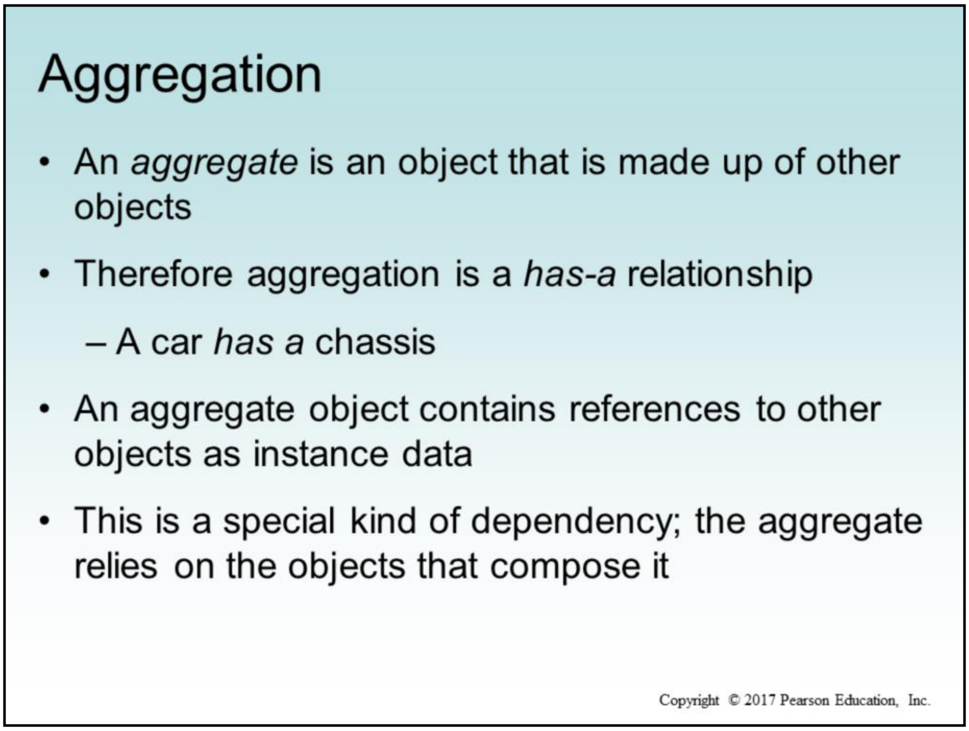
-They cannot be called by methods from other classes

-They typically exist to assist other methods in the class and are often referred to as “helper” methods



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;
   }
}
```



Aggregation

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

-Note that aggregation is a relationship we refer to as “has-a”

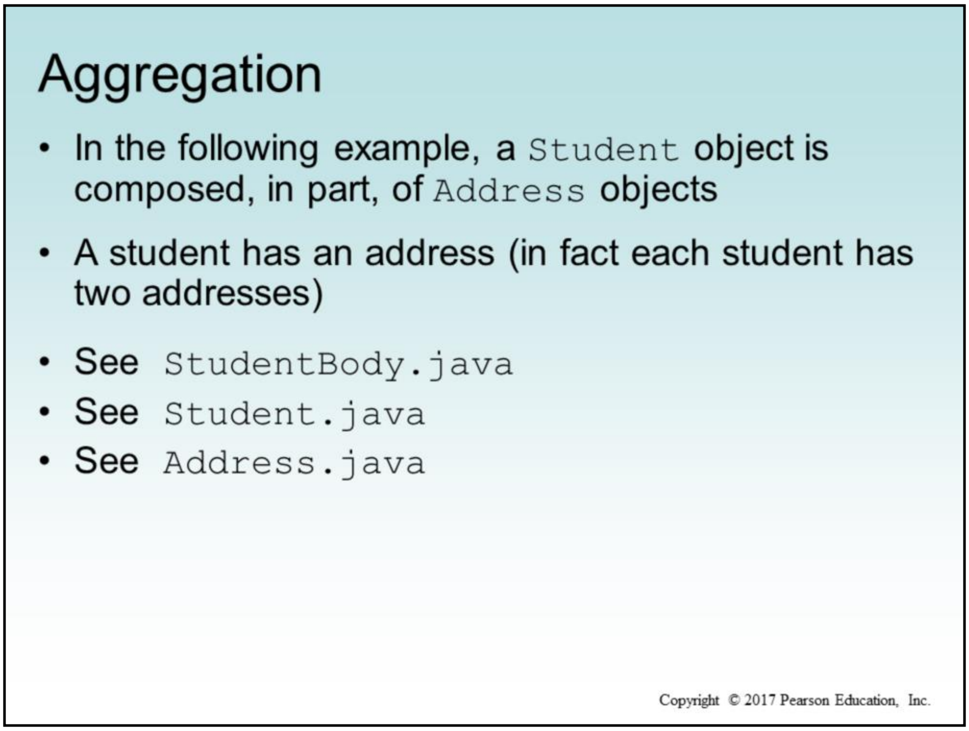
-This assists us greatly when designing classes

-If we find ourselves saying that something “has-a” something else during our design, we define an aggregate

-For example if we design a **Clock** class, we might say that a **Clock** “has a” **Alarm**

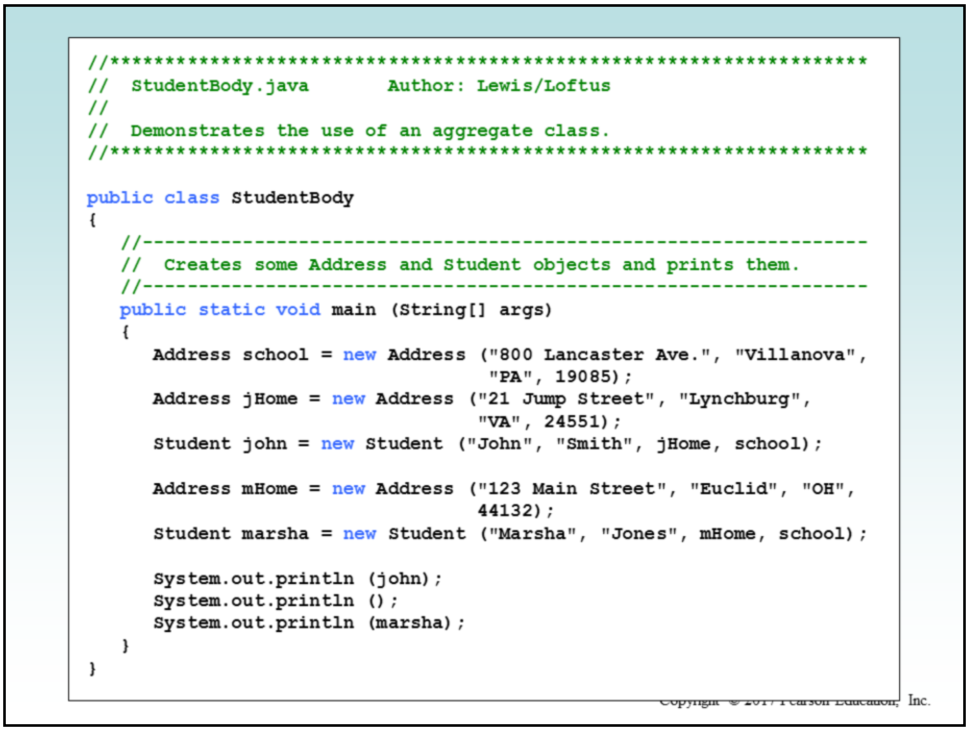
-As a result, one of the instance variables we would design in our Clock class would be an Alarm object reference variable as shown below!

```
public class Clock
{
   private Alarm alarm;
   ….
}
```



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



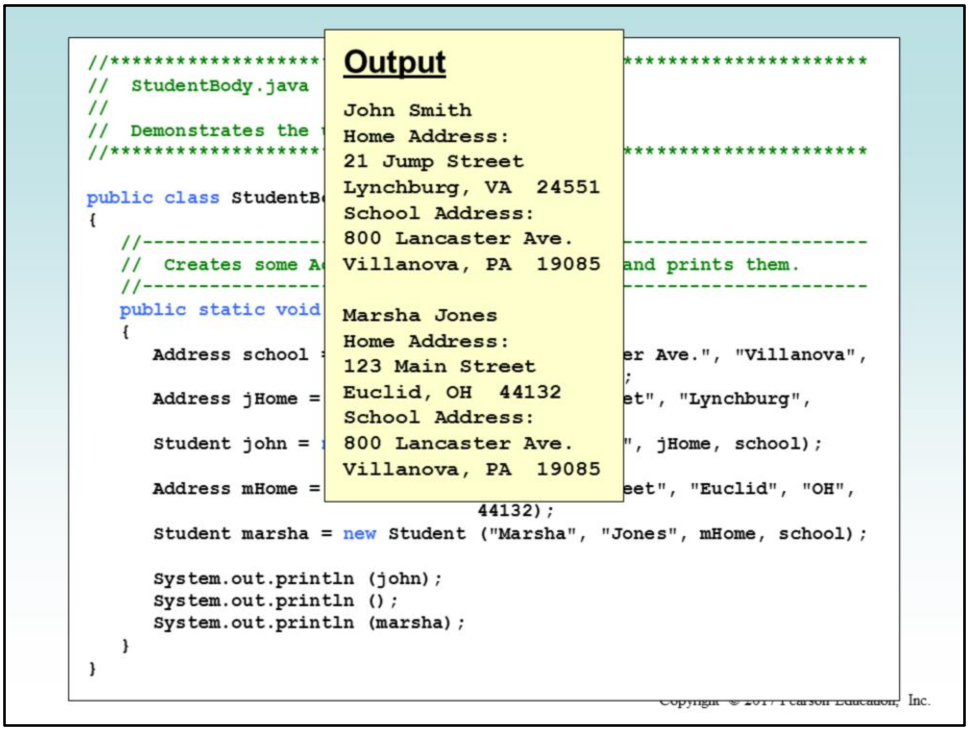
```
//********************************************************************
//  StudentBody.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an aggregate class.
//********************************************************************

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

-Note how Address objects (e.g. jHome, school) are passed as parameters to a Student constructor

-In addition, note how String objects are also passed as parameters to a Student constructor

-Remember that when a String is specified with quotes (e.g. “John”), a new String object is automatically created

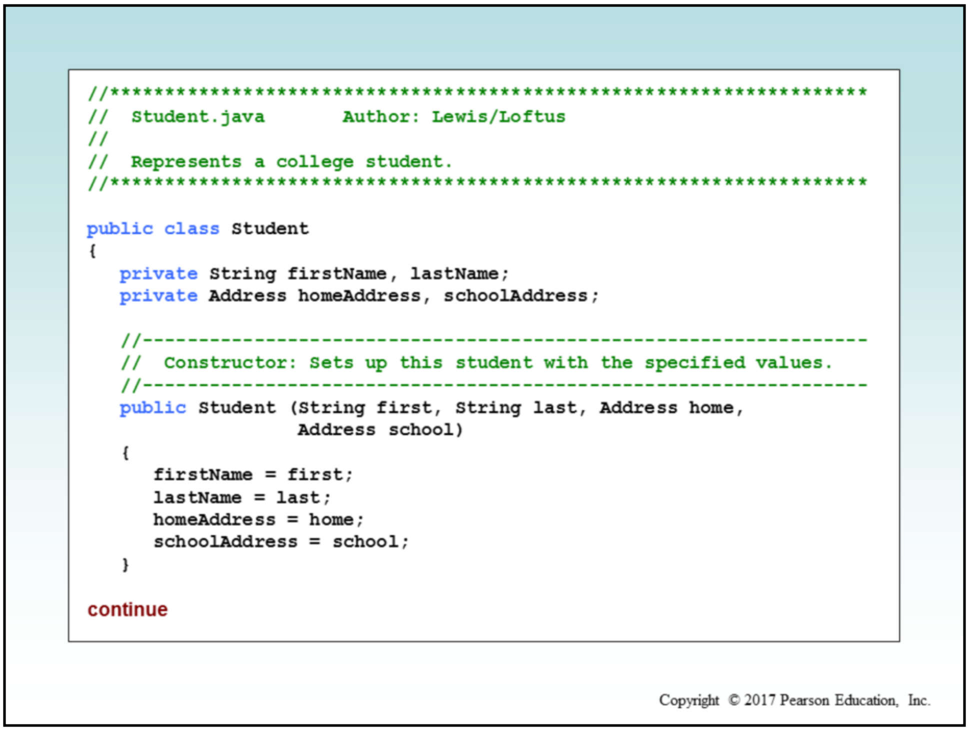
-Note when we pass a Student object to the println method, the Student’s toString method is automatically called



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
```

```
//*******************                                **********************
//  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);
   }
}
```

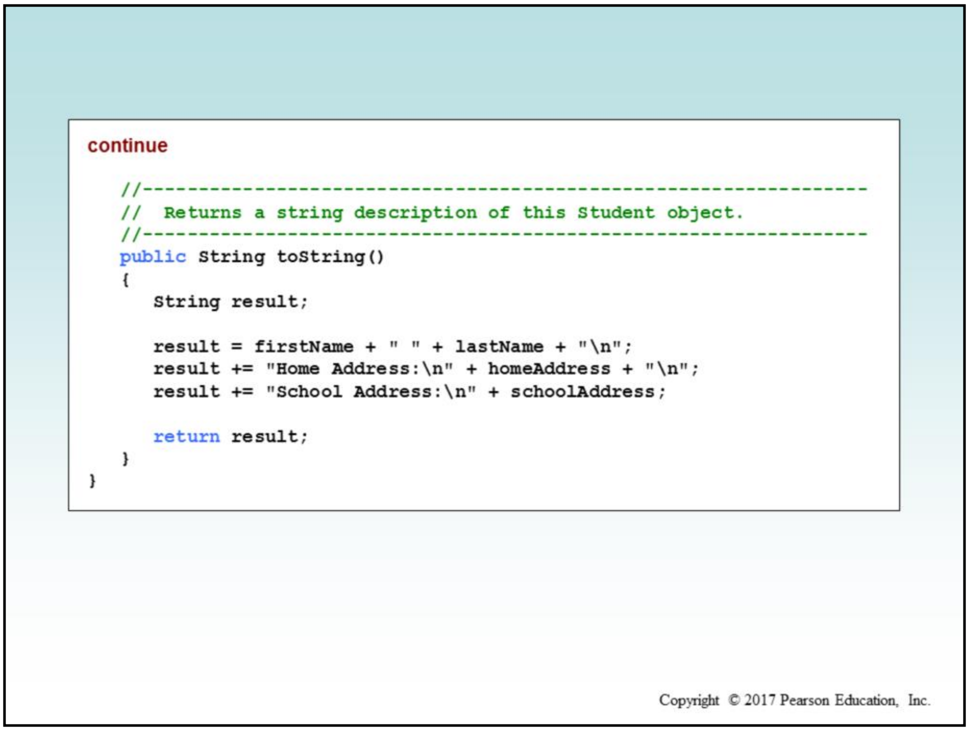


```
//********************************************************************
//  Student.java       Author: Lewis/Loftus
//
//  Represents a college student.
//********************************************************************

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;
   }
```

continue

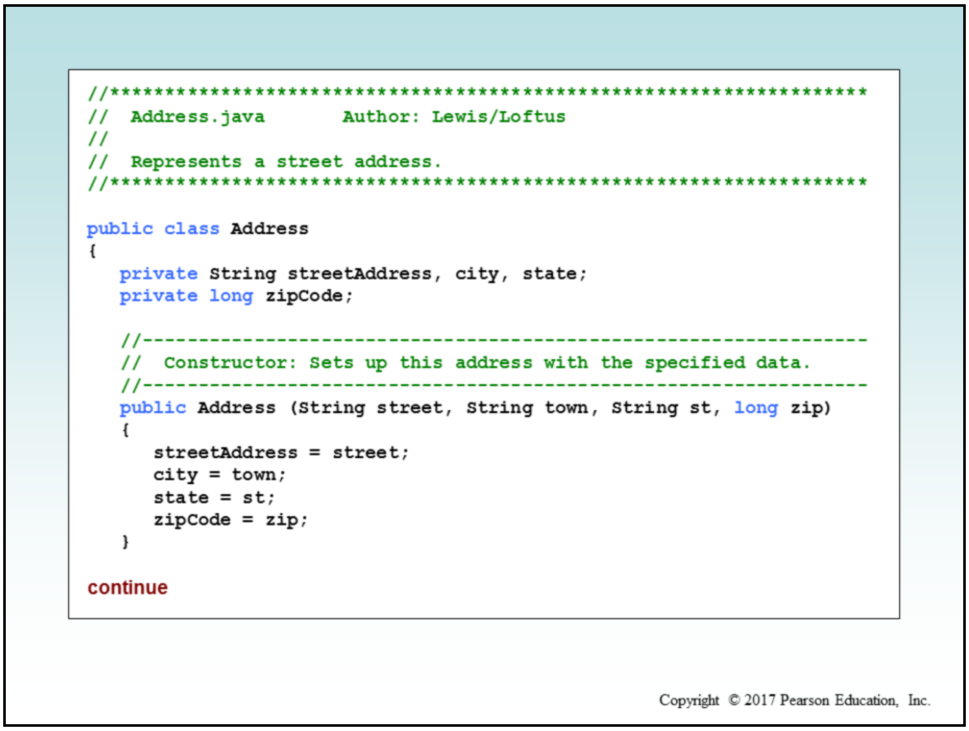
-Student is an aggregrate class because it **has** four other objects (two String and two Address objects)

-Note how the Student constructor uses String and Address objects to assign to its instance variables



continue

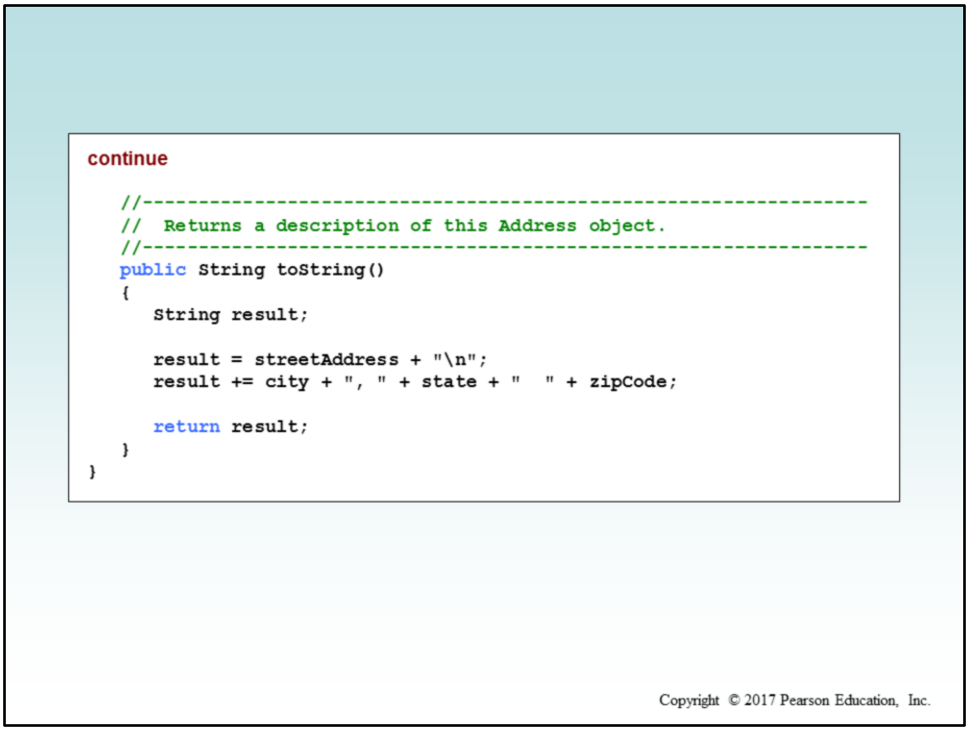
```
   //-----------------------------------------------------------------
   //  Returns a string description of this Student object.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result;

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

      return result;
   }
}
```



```
//********************************************************************
//  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;
   }
```

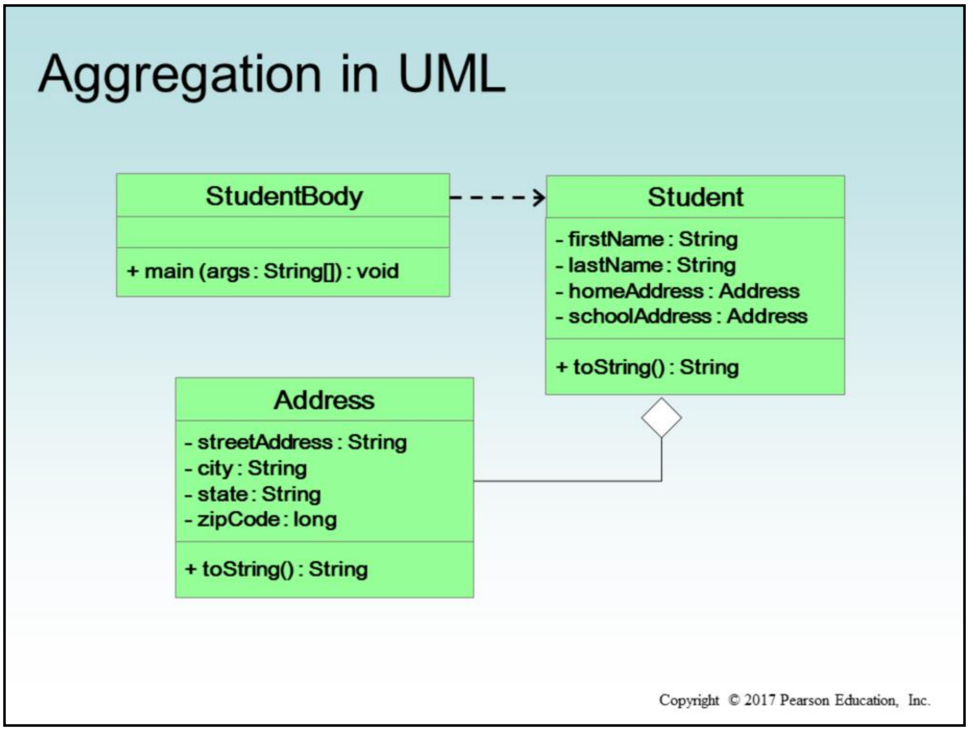
continue

-Address is also an aggregrate class because it **has** three String objects



continue

```
   //-----------------------------------------------------------------
   //  Returns a description of this Address object.
   //-----------------------------------------------------------------
   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

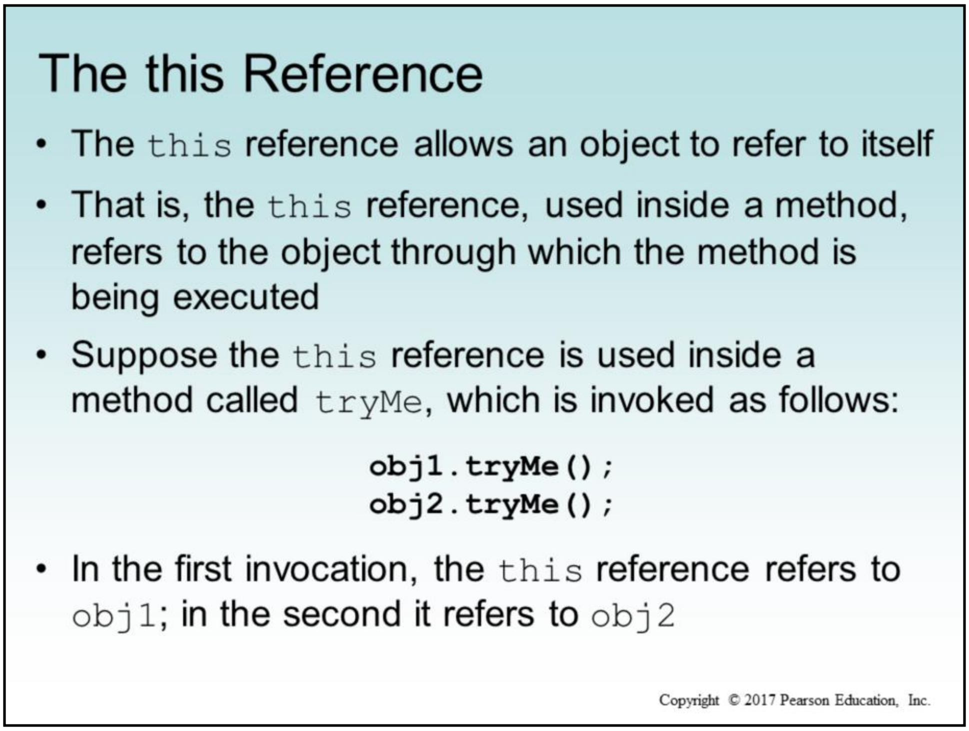
Address

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

+ toString() : String

-The open diamond indicates that a Student “has-a” Address

-The Student is an aggregate object that “has-a” Address object(s)



The this Reference

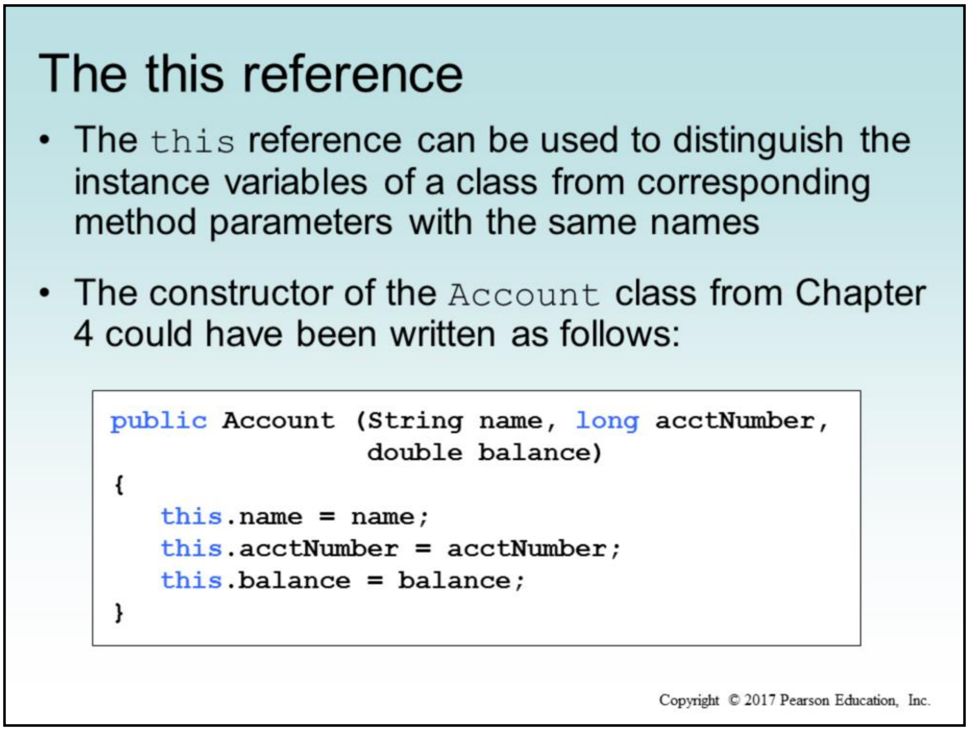
- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used inside a method called `tryMe`, which is invoked as follows:

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

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

-We can use the **this** reference variable to refer to the object itself within a method

-In other words, **this** refers to the object that called the method



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:

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

-Note how the **this** reference is used to avoid confusion between its instance variables and those passed as parameters