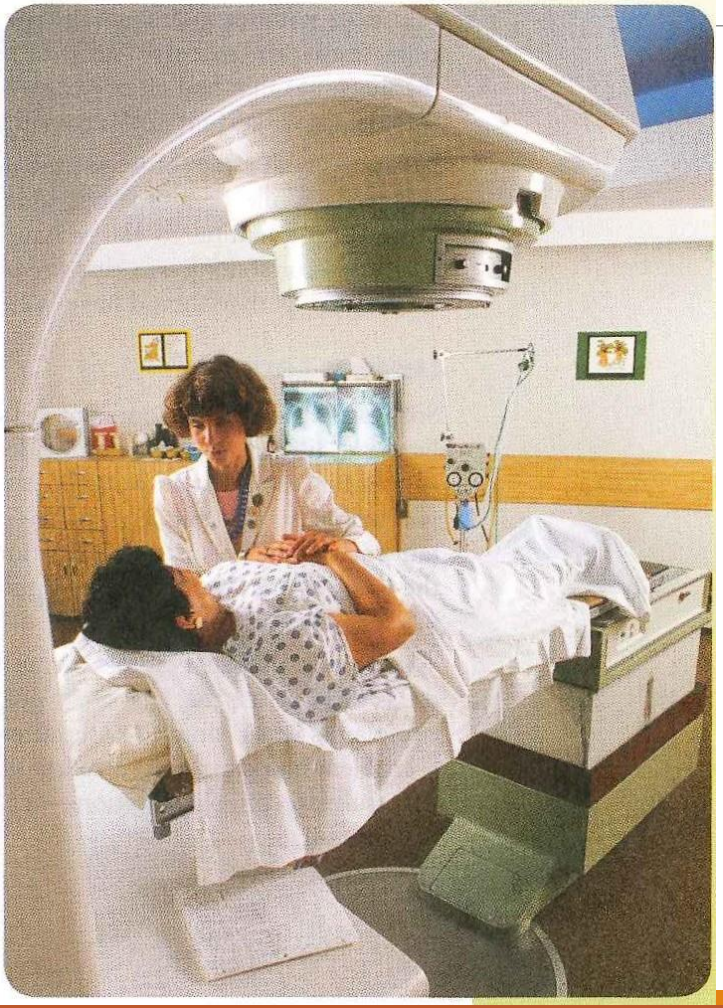
Java Class 21

Therac-25

Software errors in this radiation therapy machine caused maximum doses of radiation to be delivered if the operator mistyped a parameter and then corrected it in a certain way. For hundreds of patients, Therac-25 provided excellent treatment, but, for 6 patients, the treatment caused permanent damage.



Program Development

The creation of software involves four basic activities:

- establishing the requirements
- creating a design
- implementing the code
- testing the implementation

These activities are not strictly linear – they overlap and interact

Requirements

Software requirements specify the tasks that a program must accomplish

- what to do, not how to do it

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

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

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

Design

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

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

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

Low level design details include how individual methods will accomplish their tasks

Design

A fundamental quality of software design is high cohesion

This means that the various aspects of a software component (such as a class, object or method) fit together well

One tendency in program design is to try to cram too much functionality into one component

In good object-oriented design, each object does one thing well

Implementation

Implementation is the process of translating a design into source code

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

Almost all important decisions are made during requirements and design stages

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

Testing

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

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

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

Identifying Classes and Objects

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

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

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

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

Identifying Classes and Objects

A partial requirements document:

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

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

Identifying Classes and Objects

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

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

Examples: `Coin`, `Student`, `Message`

A class represents the concept of one such object

We are free to instantiate as many of each object as needed

Identifying Classes and Objects

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

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

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

When a class becomes too complex, it often should be decomposed into multiple smaller classes to distribute the responsibilities

Identifying Classes and Objects

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

Static Class Members

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

For example, the methods of the `Math` class are static:

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

Variables can be static as well

Determining if a method or variable should be static is an important design decision

The static Modifier

We declare static methods and variables using the `static` modifier

It associates the method or variable with the class rather than with an object of that class; that is the variable or method is not associated with any particular object of that class

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

Static Variables

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

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

All objects instantiated from the class share its static variables

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

Static Methods

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

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

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

```
//********************************************************************
//  SloganCounter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the static modifier.
//********************************************************************

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

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

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

continue

continue

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

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

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

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

continue

```
        obj = new Slo
        System.out.pr

        obj = new Slo
        System.out.pr

        obj = new Slo                                          );
        System.out.pr

        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.
Write Once, Run Anywhere.

Slogans created: 5
```

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

continue

continue

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

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

Quick Check

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

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)

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

Dependency

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

We've seen dependencies in many previous examples

We don't want numerous or complex dependencies among classes

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

A good design strikes the right balance

Dependency

Some dependencies occur between objects of the same class

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

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

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

Dependency

The following example defines a class called `RationalNumber`

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

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

```
//********************************************************************
//  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

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

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

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

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

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 engine, a chassis and wheels*

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

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

```
//********************                                    **********************
//  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
```

```
//********************************************************************
//  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

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

continue

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

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

        return result;
    }
}
```

The this Reference

The `this` reference allows you to reference the current object and allows an object to refer to itself

That is, the `this` reference, used inside a non-static method, produces the reference to the object that the method has been called for

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

The this Reference

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

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

Testing

Testing can mean many different things

It certainly includes running a completed program with various inputs

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

Some evaluations should occur before coding even begins

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

Testing

The goal of testing is to find errors

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

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

So when do we stop testing?

- Conceptual answer: Never
- Cynical answer: When we run out of time
- Better answer: When we are willing to risk that an undiscovered error still exists

Reviews

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

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

Presenting a design or code to others:

- makes us think more carefully about it
- provides an outside perspective

Reviews are sometimes called *inspections* or *walkthroughs*

Test Cases

A *test case* is a set of input and user actions, coupled with the expected results

Often test cases are organized formally into *test suites* which are stored and reused as needed

For medium and large systems, testing must be a carefully managed process

Many organizations have a separate Quality Assurance (QA) department to lead testing efforts

Defect and Regression Testing

Defect testing is the execution of test cases to uncover errors

The act of fixing an error may introduce new errors

After fixing a set of errors we should perform *regression testing* – running previous test suites to ensure new errors haven't been introduced

It is not possible to create test cases for all possible input and user actions

Therefore we should design tests to maximize their ability to find problems

Black-Box Testing

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

They are based on the input and expected output

Input can be organized into *equivalence categories*

Two input values in the same equivalence category 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

Individual Exercise

Note: this is an individual assignment!

Write an application QuizGrade that reads in and prints out a set of 20 grades into a two-dimensional array from a .txt file called studentanswers1 that represent the results from a test of 5 true-false questions for 4 students. Each row of the array represents the answers of one of the 4 students in the class. Each column represents the answers of each student to a specific question on the test. Each cell of the array contains either the Boolean value “true” or “false.”

Use the .txt file on D2L.

Some useful code:

```
File file = new File ("C://Users//Patti Ota//Documents//NetBeansProjects//JavaAssignments//src//studentAnswers2000.txt");
File file1 = new File ("C://Users//Patti Ota//Documents//NetBeansProjects//JavaAssignments//src//questionAnswers18.txt");
BufferedReader br = new BufferedReader(new FileReader(file));
BufferedReader br1 = new BufferedReader(new FileReader(file1));
```

Use a statement like the following inside a nested loop:

```
array [student][answer] = Boolean.valueOf(br.readLine());
```

Your main method should include throws IOException as in the following statement:

```
public QuizGraph() throws IOException
```

Text File I/O

- It is often useful to design programs that read input data from a text file or write output data to a text file

- Important classes for text file input (from the file):
  - BufferedReader
  - FileReader

- Important classes for text file output (to the file)
  - BufferedWriter
  - FileWriter

- Those classes are part of the java.io package, and must be imported into a program to be used: `import java.io.*;`

Read from a Text File

To read a text file `input.txt`: create a `BufferedReader` stream

```
BufferedReader reader = new BufferedReader(new
   FileReader("input.txt"));
```

You need to put the text file under the current working directory of project (i.e., the same folder as "src" and "bin") to make the above line work

If it is located somewhere else, you should use the full path name

- Full path name: "/Users/ota/java/TextIO/input.txt" (mac) or "D:\\Java\\Programs\\FileClassDemo.java" (Windows)

Methods in BufferedReader

`readLine()`: read a line into a `String`

`close()`: close a `BufferedReader` stream

A typical usage (print each line one by one):

```
String line = null;
while( (line = reader.readLine() ! = null)
     System.out.println(line);

reader.close();
```

Methods in BufferedReader

No methods to read numbers directly, so read numbers as Strings and then convert them

```
line = reader.readLine();
int i = Integer.parseInt(line);

line = reader.readLine();
double d = Double.parseDouble (line);

line = reader.readLine();
boolean b = Boolean.parseBoolean(line);
```

Write to a Text File

To write to a text file `output.txt`: create a `BufferedWriter` stream

```
BufferedWriter writer = new BufferedWriter(new
      FileWriter("output.txt"));
```

"output.txt" will be written to the current working directory of project (i.e., the same folder as "src" and "bin")

Can use full name to specify the destination as well

If a file with the same name already exists, it will be overwritten (data in the original file is lost)

Methods in BufferedWritter

`write(String s)`: write a `String s` to the file (without a new line)

`newLine()` : write a new line to the file

`close()`: close a `BufferedReader` stream

No methods to write numbers directly, so need to convert them into Strings before calling the method `write`

```
int i = 10;
writer.write(Integer.toString(i));
writer.newLine();
double d = 20.2;
writer.write(Double.toString(d));
```

Assignment for Class 22

Review SloganCounter, Slogan, RationalTester, RationalNumber, StudentBody, Student, Address,

Read Chapter 7.5, 7.6, 7.7, 7.8