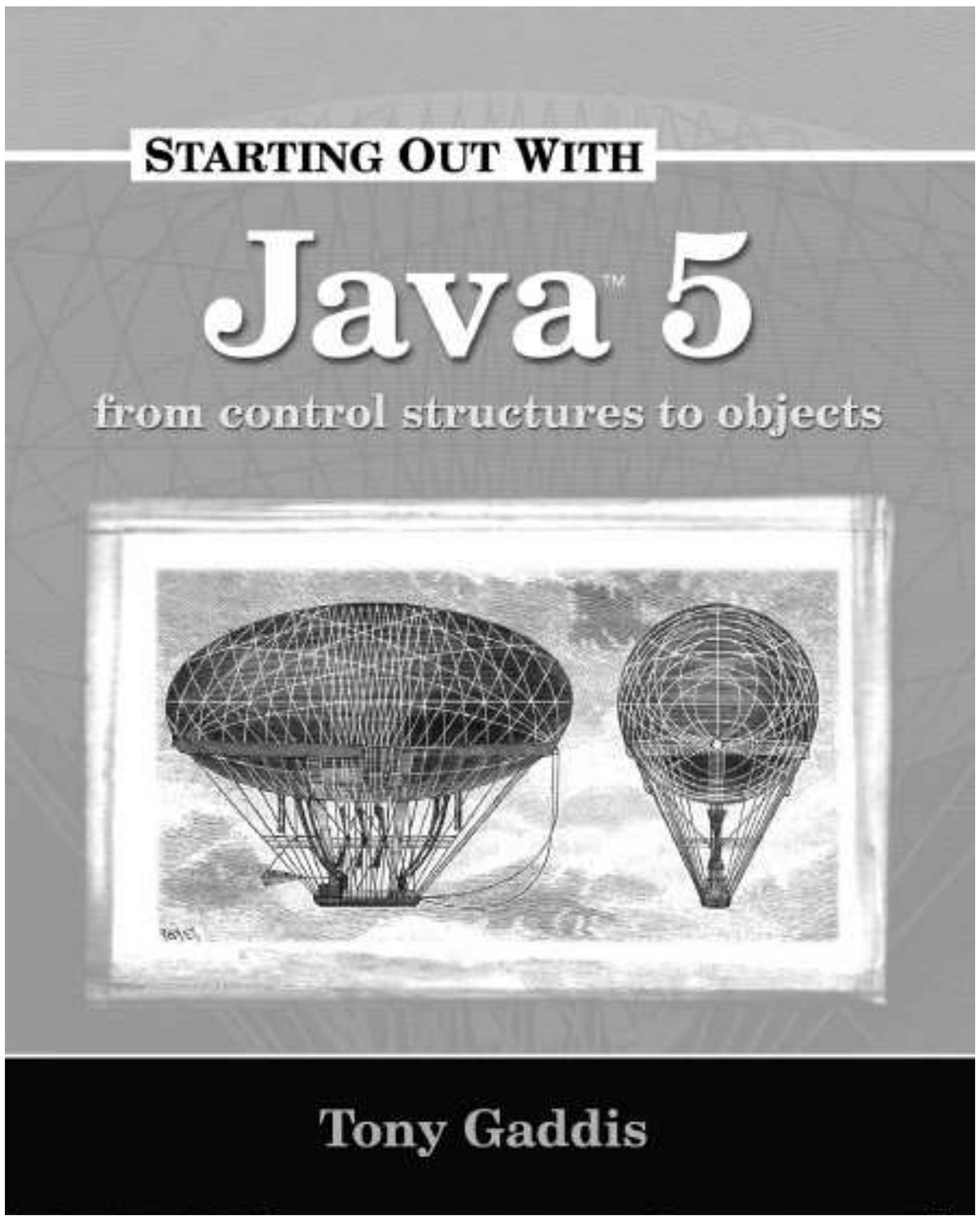
Lab Manual

to Accompany



Diane Christie

University of Wisconsin – Stout

PEARSON
Addison Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Publisher	*Greg Tobin*
Senior Acquisitions Editor	*Michael Hirsch*
Editorial Assistant	*Lindsey Triebel*
Cover Designer	*Nicole Clayton*
Marketing Manager	*Michelle Brown*
Marketing Assistant	*Dana Lopreato*
Prepress and Manufacturing	*Caroline Fell*
Supplement Coordination	*Marianne Groth*
Proofreader	*Melanie Aswell*

ISBN 0-321-43424-2
1 2 3 4 5 6 7 8 9 10—BB—09 08 07 06

Preface

About this Lab Manual

This lab manual accompanies *Starting Out With Java 5: From Control Structures to Objects*, by Tony Gaddis. Each lab gives students hands on experience with the major topics in each chapter. It is designed for closed laboratories—regularly scheduled classes supervised by an instructor, with a length of approximately two hours. Lab manual chapters correspond to textbook chapters. Each chapter in the lab manual contains learning objectives, an introduction, one or two projects with various tasks for the students to complete, and a listing of the code provided as the starting basis for each lab. Labs are intended to be completed after studying the corresponding textbook chapter, but prior to programming challenges for each chapter in the textbook.

Students should copy the partially written code (available at www.aw.com/cssupport) and use the instructions provided in each task to complete the code so that it is operational. Instructions will guide them through each lab having them add code at specified locations in the partially written program. Students will gain experience in writing code, compiling and debugging, writing testing plans, and finally executing and testing their programs.

Note: Labs 7 and 12 are written entirely by the student using the instructions in the various tasks, so there is no code provided as a starting basis.

What You'll Find in this Lab Manual

The Lab Manual contains 15 labs that help students learn how to apply introductory programming concepts:

• **Chapter 1 Lab**	Algorithms, Errors, and Testing
• **Chapter 2 Lab**	Java Fundamentals
• **Chapter 3 Lab**	Selection Control Structures
• **Chapter 4 Lab**	Loops and Files
• **Chapter 5 Lab**	Methods
• **Chapter 6 Lab**	Classes and Objects
• **Chapter 7 Lab**	GUI Applications
• **Chapter 8 Lab**	Arrays
• **Chapter 9 Lab**	More Classes and Objects
• **Chapter 10 Lab**	Text Processing and Wrapper Classes
• **Chapter 11 Lab**	Inheritance
• **Chapter 12 Lab**	Exceptions and I/O Streams
• **Chapter 13 Lab**	Advanced GUI Applications
• **Chapter 14 Lab**	Applets and More
• **Chapter 15 Lab**	Recursion

Supplementary Materials

- Students can find source code files for the labs at www.aw.com/cssupport, under author "Christie" and title "*Lab Manual to Accompany Starting Out with Java 5: From Control Structures to Objects*" or "Gaddis", "*Starting Out with Java 5: From Control Structures to Objects*."
- Solution files and source code are available to qualified instructors at Addison-Wesley's Instructor Resource Center. Register at www.aw.com/irc and search for author "Gaddis."

Acknowledgements

I would like to thank everyone at Addison-Wesley for making this lab manual a reality, Tony Gaddis for having the confidence in me to write labs to accompany his books and my colleagues who have contributed ideas to help develop these labs.

I also thank my students at the University of Wisconsin-Stout for giving me feedback on these labs to continue to improve them.

Most of all, I want to thank my family: Michael, Andrew, and Pamela for all of their encouragement, patience, love, and support.

Contents

Chapter 1 Lab
Algorithms, Errors, and Testing

Objectives

- Be able to write an algorithm
- Be able to compile a Java program
- Be able to execute a Java program using the Sun JDK or a Java IDE
- Be able to test a program
- Be able to debug a program with syntax and logic errors

Introduction

Your teacher will introduce your computer lab and the environment you will be using for programming in Java.

In chapter 1 of the textbook, we discuss writing your first program. The example calculates the user's gross pay. It calculates the gross pay by multiplying the number of hours worked by hourly pay rate. However, it is not always calculated this way. What if you work 45 hours in a week? The hours that you worked over 40 hours are considered overtime. You will need to be paid time and a half for the overtime hours you worked.

In this lab, you are given a program which calculates user's gross pay with or without overtime. You are to work backwards this time, and use pseudocode to write an algorithm from the Java code. This will give you practice with algorithms while allowing you to explore and understand a little Java code before we begin learning the Java programming language.

You will also need to test out this program to ensure the correctness of the algorithm and code. You will need to develop test data that will represent all possible kinds of data that the user may enter.

You will also be debugging a program. There are several types of errors. In this lab, you will encounter syntax and logic errors. We will explore runtime errors in lab 2.

1. Syntax Errors—errors in the "grammar" of the programming language. These are caught by the compiler and listed out with line number and error found. You will learn how to understand what they tell you with experience. All syntax errors must be corrected before the program will run. If the program runs, this

does not mean that it is correct, only that there are no syntax errors. Examples of syntax errors are spelling mistakes in variable names, missing semicolon, unpaired curly braces, etc.

2. Logic Errors—errors in the logic of the algorithm. These errors emphasize the need for a correct algorithm. If the statements are out of order, if there are errors in a formula, or if there are missing steps, the program can still run and give you output, but it may be the wrong output. Since there is no list of errors for logic errors, you may not realize you have errors unless you check your output. It is very important to know what output you expect. You should test your programs with different inputs, and know what output to expect in each case. For example, if your program calculates your pay, you should check three different cases: less than 40 hours, 40 hours, and more than 40 hours. Calculate each case by hand before running your program so that you know what to expect. You may get a correct answer for one case, but not for another case. This will help you figure out where your logic errors are.

3. Run time errors—errors that do not occur until the program is run, and then may only occur with some data. These errors emphasize the need for completely testing your program.

Task #1 Writing an Algorithm

1. Copy the file *Pay.java* (see code listing 1.1) from www.aw.com/cssupport or as directed by your instructor.
2. Open the file in your Java Integrated Development Environment (IDE) or a text editor as directed by your instructor. Examine the file, and compare it with the detailed version of the pseudocode in step number 3, section 1.6 of the textbook. Notice that the pseudocode does not include every line of code. The program code includes identifier declarations and a statement that is needed to enable Java to read from the keyboard. These are not part of actually completing the task of calculating pay, so they are not included in the pseudocode. The only important difference between the example pseudocode and the Java code is in the calculation. Below is the detailed pseudocode from the example, but without the calculation part. You need to fill in lines that tell in English what the calculation part of *Pay.java* is doing.

```
Display   How many hours did you work?
Input     hours
Display   How much do you get paid per hour?
Input     rate
```

Display the value in the pay variable.

Task #2 Compile and Execute a Program

1. Compile the *Pay.java* using the Sun JDK or a Java IDE as directed by your instructor.
2. You should not receive any error messages.
3. When this program is executed, it will ask the user for input. You should calculate several different cases by hand. Since there is a critical point at which the calculation changes, you should test three different cases: the critical point, a number above the critical point, and a number below the critical point. You want to calculate by hand so that you can check the logic of the program. Fill in the chart below with your test cases and the result you get when calculating by hand.
4. Execute the program using your first set of data. Record your result. You will need to execute the program three times to test all your data. Note: you do not need to compile again. Once the program compiles correctly once, it can be executed many times. You only need to compile again if you make changes to the code.

Hours	**Rate**	**Pay (hand calculated)**	**Pay (program result)**

Task #3 Debugging a Java Program

1. Copy the file *SalesTax.java* (see code listing 1.2) from www.aw.com/cssupport or as directed by your instructor.
2. Open the file in your IDE or text editor as directed by your instructor. This file contains a simple Java program that contains errors. Compile the program. You should get a listing of syntax errors. Correct all the syntax errors, you may want to recompile after you fix some of the errors.
3. When all syntax errors are corrected, the program should compile. As in the previous exercise, you need to develop some test data. Use the chart below to record your test data and results when calculated by hand.
4. Execute the program using your test data and recording the results. If the output of the program is different from what you calculated, this usually indicates a logic error. Examine the program and correct logic error. Compile the program and execute using the test data again. Repeat until all output matches what is expected.

Item	**Price**	**Tax**	**Total (calculated)**	**Total (output)**

Code Listing 1.1 (Pay.java)

```
//This program calculates the user's gross pay

import java.util.Scanner;   //to be able to read from the keyboard

public class Pay
{
     public static void main(String [] args)
     {
          //create a Scanner object to read from the keyboard
          Scanner keyboard = new Scanner(System.in);

          //identifier declarations
          double hours;      //number of hours worked
          double rate;       //hourly pay rate
          double pay;        //gross pay

          //display prompts and get input
          System.out.print("How many hours did you work? ");
          hours = keyboard.nextDouble();
          System.out.print("How much do you get paid per hour? ");
          rate = keyboard.nextDouble();

          //calculations
          if(hours <= 40)
               pay = hours * rate;
          else
               pay = (hours - 40) * (1.5 * rate) + 40 * rate;

          //display results
          System.out.println("You earned $" + pay);
     }
}
```

Code Listing 1.2 (SalesTax.java)

```
//This program calculates the total price which includes sales //tax

import java.util.Scanner;

public class SalesTax
{
     public static void main(String[] args)
     {
          //identifier declarations
          final double TAX_RATE = 0.055;
          double price;
          double tax
          double total;
          String item;

          //create a Scanner object to read from the keyboard
          Scanner keyboard = new Scanner(System.in);

          //display prompts and get input
          System.out.print("Item description:  ");
          item = keyboard.nextLine();
          System.out.print("Item price:  $");
          price = keyboard.nextDouble();

          //calculations
          tax = price + TAX_RATE;
          total = price * tax;

          //display results
          System.out.print(item + "       $");
          System.out.println(price);
          System.out.print("Tax          $");
          System.out.println(tax);
          System.out.print("Total       $");
          System.out.println(total);
     }
}
```

Chapter 2 Lab
Java Fundamentals

Objectives

- Write arithmetic expressions to accomplish a task
- Use casting to convert between primitive types
- Use a value-returning library method and a library constant
- Use string methods to manipulate string data
- Communicate with the user by using the Scanner class or dialog boxes
- Create a program from scratch by translating a pseudocode algorithm
- Be able to document a program

Introduction

This lab is designed to give you practice with some of the basics in Java. We will continue ideas from lab 1 by correcting logic errors while looking at mathematical formulas in Java. We will explore the difference between integer division and division on your calculator as well as reviewing the order of operations.

We will also learn how to use mathematical formulas that are preprogrammed in Java. On your calculator there are buttons to be able to do certain operations, such as raise a number to a power or use the number pi. Similarly, in Java, we will have programs that are available for our use that will also do these operations. Mathematical operations that can be performed with the touch of a button on a calculator are also available in the Math class. We will learn how to use a Math class method to cube the radius in the formula for finding the volume of a sphere.

This lab also introduces communicating with the user. We have already seen how console input and output work in lab 1. We will now need to learn how to program user input, by investigating the lines of code that we need to add in order to use the Scanner class. We will also learn the method call needed for output.

Alternately, you may use dialog boxes for communicating with the user. An introduction to graphical user interface (GUI) programming is explored using the JOptionPane class.

The String class is introduced and we will use some of the available methods to prepare you for string processing.

We will bring everything we have learned together by creating a program from an algorithm. Finally, you will document the program by adding comments. Comments are not read by the computer, they are for use by the programmer. They are to help a programmer document what the program does and how it accomplishes it. This is very important when a programmer needs to modify code that is written by another person.

Task #1 Correcting Logic Errors in Formulas

1. Download the file *NumericTypes.java* (see code listing 2.1) from www.aw.com/cssupport or as directed by your instructor.
2. Compile the source file, run the program, and observe the output. Some of the output is ***incorrect***. You need to **correct logic errors** in the average formula and the temperature conversion formula. The logic errors could be due to conversion between data types, order of operations, or formula problems. The necessary formulas are

$$average = \frac{score1 + score2}{numberOfScores} \qquad \mathrm{C} = \tfrac{5}{9}(F - 32)$$

3. Each time you make changes to the program code, you must compile again for the changes to take effect before running the program again.
4. Make sure that the output makes sense before you continue. The average of 95 and 100 should be 97.5 and the temperature that water boils is 100 degrees Celsius.

Task #2 Using the Scanner Class for User Input

1. Add an import statement above the class declaration to make the Scanner class available to your program.
2. In the main method, create a Scanner object and connect it to the System.in object.
3. Prompt the user to enter his/her first name.
4. Read the name from the keyboard using the nextLine method, and store it into a variable called `firstName` (you will need to declare any variables you use).
5. Prompt the user to enter his/her last name.
6. Read the name from the keyboard and store it in a variable called `lastName`.
7. Concatenate the `firstName` and `lastName` with a space between them and store the result in a variable called `fullName`.
8. Print out the `fullName`.
9. Compile, debug, and run, using your name as test data.
10. Since we are adding on to the same program, each time we run the program we will get the output from the previous tasks before the output of the current task.

Task #2 (Alternate) Using Dialog Boxes for User Input

1. Add an import statement above the class declaration to make the JOptionPane class available to your program.
2. In the main method, prompt the user to enter his/her first name by displaying an input dialog box and storing the user input in a variable called `firstName` (you will need to declare any variables you use).
3. Prompt the user to enter his/her last name by displaying an input dialog box and storing the user input in a variable called `lastName`.
4. Concatenate the `firstName` and `lastName` with a space between them and store the result in a variable called `fullName`.
5. Display the `fullName` using a message dialog box.
6. Compile, debug, and run, using your name as test data.
7. Since we are adding on to the same program, each time we run the program we will get the output from the previous tasks before the output of the current task.

Task #3 Working with Strings

1. Use the **charAt** method to get the first character in `firstName` and store it in a variable called `firstInitial` (you will need to declare any variables that you use).
2. Print out the user's first initial.
3. Use the to**UpperCase** method to change the `fullName` to all capitals and store it back into the `fullName` variable
4. Add a line that prints out the value of `fullName` and how many characters (including the space) are in the string stored in `fullName` (use the method **length** to obtain that information).
5. Compile, debug, and run. The new output added on after the output from the previous tasks should have your initials and your full name in all capital letters.

Task #4 Using Predefined Math Functions

1. Add a line that prompts the user to enter the diameter of a sphere.
2. Read in and store the number into a variable called diameter (you will need to declare any variables that you use).
3. The diameter is twice as long as the radius, so calculate and store the radius in an appropriately named variable.
4. The formula for the volume of a sphere is

   $$V = \frac{4}{3}\pi r^3$$

   Convert the formula to Java and add a line which calculates and stores the value of volume in an appropriately named variable. Use Math.PI for π and Math.pow to cube the radius.
5. Print your results to the screen with an appropriate message.
6. Compile, debug, and run using the following test data and record the results.

Diameter	**Volume (hand calculated)**	**Volume (resulting output)**
2		
25.4		
875,000		

Task #5 Create a program from scratch

In this task the student will create a new program that calculates gas mileage in miles per gallon. The student will use string expressions, assignment statements, input and output statements to communicate with the user.

1. Create a new file in your IDE or text editor.
2. Create the shell for your first program by entering:

```
public class Mileage
{
     public static void main(String[] args)
     {
          // add your declaration and code here
     }
}
```

3. Save the file as *Mileage.java*.
4. Translate the algorithm below into Java. Don't forget to declare variables before they are used. Each variable must be one word only (no spaces).

```
Print a line indicating this program will calculate mileage
Print prompt to user asking for miles driven
Read in miles driven
Print prompt to user asking for gallons used
Read in gallons used
Calculate miles per gallon by dividing miles driven by gallons used
Print miles per gallon along with appropriate labels
```

5. Compile the program and debug, repeating until it compiles successfully.
6. Run the program and test it using the following sets of data and record the results:

Miles driven	**Gallons used**	**Miles per gallon (hand calculated)**	**Miles per gallon (resulting output)**
2000	100		
500	25.5		
241.5	10		
100	0		

7. The last set of data caused the computer to divide 100 by 0, which resulted in what is called a **runtime error**. Notice that runtime can occur on programs which compile and run on many other sets of data. This emphasizes the need to thoroughly test you program with all possible kinds of data.

Task #6 Documenting a Java Program

1. Compare the code listings of *NumericTypes.java* with *Mileage.java*. You will see that *NumericTypes.java* has lines which have information about what the program is doing. These lines are called comments and are designated by the `//` at the beginning of the line. Any comment that starts with `/**` and ends with `*/` is considered a documentation comment. These are typically written just before a class header, giving a brief description of the class. They are also used for documenting methods in the same way.
2. Write a documentation comment at the top of the program which indicates the purpose of the program, your name, and today's date.
3. Add comment lines after each variable declaration, indicating what each variable represents.
4. Add comment lines for each section of the program, indicating what is done in that section.
5. Finally add a comment line indicating the purpose of the calculation.

Code Listing 2.1 (NumericTypes.java)

```
/**
   This program demonstrates how numeric types and operators behave
*/

//TASK #2 Add import statement here to use the Scanner class
//TASK #2  (Alternate) Add import statment to use JOptionPane
//class

public class NumericTypes
{
      public static void main (String [] args)
      {
            //TASK #2 Create a Scanner object here
            //(not used for alternate)

            //identifier declarations
            final int NUMBER = 2 ;    // number of scores
            final int SCORE1 = 100;   // first test score
            final int SCORE2 = 95;    // second test score
            final int BOILING_IN_F = 212; // freezing temperature
            int fToC;                 // temperature in Celsius
            double average;          // arithmetic average
            String output;      // line of output to print out
            //TASK #2 declare variables used here
            //TASK #3 declare variables used here
            //TASK #4 declare variables used here

            // Find an arithmetic average
            average = SCORE1 + SCORE2 / NUMBER;
            output = SCORE1 + " and " + SCORE2 +
                  " have an average of " + average;
            System.out.println(output);

            // Convert Fahrenheit temperatures to Celsius
            fToC = 5/9 * (BOILING_IN_F - 32);
            output = BOILING_IN_F + " in Fahrenheit is " + fToC
                  + " in Celsius.";
```

Code Listing 2.1 continued on next page.

```
        System.out.println(output);
        System.out.println();       // to leave a blank line

        // ADD LINES FOR TASK #2 HERE
        // prompt the user for first name
        // read the user’s first name
        // prompt the user for last name
        // read the user’s last name
        // concatenate the user’s first and last names
        // print out the user’s full name

        System.out.println();       // to leave a blank line

        // ADD LINES FOR TASK #3 HERE
        // get the first character from the user’s first name
        // print out the user’s first initial
        // convert the user’s full name to all capital letters
        // print out the user’s full name in all capital
        // letters and the number of characters in it

        System.out.println();       // to leave a blank line

        // ADD LINES FOR TASK #4 HERE
        // prompt the user for a diameter of a sphere
        // read the diameter
        // calculate the radius
        // calculate the volume
        // print out the volume
    }
}
```

Chapter 3 Lab
Selection Control Structures

Objectives

- Be able to construct boolean expressions to evaluate a given condition
- Be able to compare Strings
- Be able to use a flag
- Be able to construct if and if-else-if statements to perform a specific task
- Be able to construct a switch statement
- Be able to format numbers

Introduction

Up to this point, all the programs you have had a sequential control structure. This means that all statements are executed in order, one after another. Sometimes we need to let the computer make decisions, based on the data. A selection control structure allows the computer to select which statement to execute.

In order to have the computer make a decision, it needs to do a comparison. So we will work with writing boolean expressions. Boolean expressions use relational operators and logical operators to create a condition that can be evaluated as true or false.

Once we have a condition, we can conditionally execute statements. This means that there are statements in the program that may or may not be executed, depending on the condition.

We can also chain conditional statements together to allow the computer to choose from several courses of action. We will explore this using nested if-else statements as well as a switch statement.

In this lab, we will be editing a pizza ordering program. It creates a Pizza object to the specifications that the user desires. It walks the user through ordering, giving the user choices, which the program then uses to decide how to make the pizza and how much the cost of the pizza will be. The user will also receive a $2.00 discount if his/her name is Mike or Diane.

Task #1 The if Statement, Comparing Strings, and Flags

1. Copy the file *PizzaOrder.java* (see code listing 3.1) from www.aw.com/cssupport or as directed by your instructor.
2. Compile and run *PizzaOrder.java*. You will be able to make selections, but at this point, you will always get a Hand-tossed pizza at a base cost of $12.99 no matter what you select, but you will be able to choose toppings, and they should add into the price correctly. You will also notice that the output does not look like money. So we need to edit *PizzaOrder.java* to complete the program so that it works correctly.
3. Construct a simple if statement. The condition will compare the String input by the user as his/her first name with the first names of the owners, Mike and Diane. Be sure that the comparison is not case sensitive.
4. If the user has either first name, set the discount flag to true. This will not affect the price at this point yet.

Task #2 The if-else-if Statement

1. Write an if-else-if statement that lets the computer choose which statements to execute by the user input size (10, 12, 14, or 16). For each option, the cost needs to be set to the appropriate amount.
2. The default else of the above if-else-if statement should print a statement that the user input was not one of the choices, so a 12 inch pizza will be made. It should also set the size to 12 and the cost to 12.99.
3. Compile, debug, and run. You should now be able to get correct output for size and price (it will still have Hand-tossed crust, the output won't look like money, and no discount will be applied yet). Run your program multiple times ordering a 10, 12, 14, 16, and 17 inch pizza.

Task #3 Switch Statement

1. Write a switch statement that compares the user's choice with the appropriate characters (make sure that both capital letters and small letters will work).
2. Each case will assign the appropriate string indicating crust type to the crust variable.
3. The default case will print a statement that the user input was not one of the choices, so a Hand-tossed crust will be made.
4. Compile, debug, and run. You should now be able to get crust types other than Hand-tossed. Run your program multiple times to make sure all cases of the switch statement operate correctly.

Task #4 Using a Flag as a Condition

1. Write an if statement that uses the flag as the condition. Remember that the flag is a Boolean variable, therefore is true or false. It does not have to be compared to anything.
2. The body of the if statement should contain two statements:
   a) A statement that prints a message indicating that the user is eligible for a $2.00 discount.
   b) A statement that reduces the variable cost by 2.
3. Compile, debug, and run. Test your program using the owners' names (both capitalized and not) as well as a different name. The discount should be correctly at this time.

Task #5 Formatting Numbers

1. Add an import statement to use the DecimalFormat class as indicated above the class declaration.
2. Create a DecimalFormat object that always shows 2 decimal places.
3. Edit the appropriate lines in the main method so that any monetary output has 2 decimal places.
4. Compile, debug, and run. Your output should be completely correct at this time, and numeric output should look like money.

Code Listing 3.1 (PizzaOrder.java)

```
/**
   This program allows the user to order a pizza
*/

import java.util.Scanner;
import java.text.DecimalFormat;

public class PizzaOrder
{
   public static void main (String [] args)
   {
      //TASK #5 Create a DecimalFormat object with 2 decimal
      //places

      //Create a Scanner object to read input
      Scanner keyboard = new Scanner (System.in);

      String firstName;                //user's first name
      boolean discount = false;        //flag, true if user is
                                       //eligible for discount
      int inches;                      //size of the pizza
      char crustType;                  //code for type of crust
      String crust = "Hand-tossed";    //name of crust
      double cost = 12.99;             //cost of the pizza
      final double TAX_RATE = .08;     //sales tax rate
      double tax;                      //amount of tax
      char choice;                     //user's choice
      String input;                    //user input
      String toppings = "Cheese ";     //list of toppings
      int numberOfToppings = 0;        //number of toppings

      //prompt user and get first name
      System.out.println("Welcome to Mike and Diane's Pizza");
      System.out.print("Enter your first name:  ");
      firstName = keyboard.nextLine();
```

Code Listing 3.1 continued on next page.

```
//determine if user is eligible for discount by
//having the same first name as one of the owners
//ADD LINES HERE FOR TASK #1

//prompt user and get pizza size choice
System.out.println("Pizza Size (inches)   Cost");
System.out.println("      10          $10.99”);
System.out.println("      12          $12.99");
System.out.println("      14          $14.99");
System.out.println("      16          $16.99");
System.out.println("What size pizza would you like?");
System.out.print(
   "10, 12, 14, or 16 (enter the number only): ");
inches = keyboard.nextInt();

//set price and size of pizza ordered
//ADD LINES HERE FOR TASK #2

//consume the remaining newline character
keyboard.nextLine();

//prompt user and get crust choice
System.out.println("What type of crust do you want? ");
System.out.print("(H)Hand-tossed, (T) Thin-crust, or " +
      "(D) Deep-dish (enter H, T, or D): ");
input = keyboard.nextLine();
crustType = input.charAt(0);

//set user’s crust choice on pizza ordered
//ADD LINES FOR TASK #3

//prompt user and get topping choices one at a time

System.out.println("All pizzas come with cheese.");
System.out.println("Additional toppings are $1.25 each,"
      + " choose from");
System.out.println("Pepperoni, Sausage, Onion, Mushroom");
```

Code Listing 3.1 continued on next page.

```
//if topping is desired,
//add to topping list and number of toppings
System.out.print("Do you want Pepperoni?  (Y/N):  ");
input = keyboard.nextLine();
choice = input.charAt(0);
if (choice == 'Y' || choice == 'y')
{
   numberOfToppings += 1;
   toppings = toppings + "Pepperoni ";
}
System.out.print("Do you want Sausage?  (Y/N):  ");
input = keyboard.nextLine();
choice = input.charAt(0);
if (choice == 'Y' || choice == 'y')
{
   numberOfToppings += 1;
   toppings = toppings + "Sausage ";
}
System.out.print("Do you want Onion?  (Y/N):  ");
input = keyboard.nextLine();
choice = input.charAt(0);
if (choice == 'Y' || choice == 'y')
{
   numberOfToppings += 1;
   toppings = toppings + "Onion ";
}
System.out.print("Do you want Mushroom?  (Y/N):  ");
input = keyboard.nextLine();
choice = input.charAt(0);
if (choice == 'Y' || choice == 'y')
{
   numberOfToppings += 1;
   toppings = toppings + "Mushroom ";
}

//add additional toppings cost to cost of pizza
```

Code Listing 3.1 continued on next page.

```
        cost = cost + (1.25*numberOfToppings);

        //display order confirmation
        System.out.println();
        System.out.println("Your order is as follows: ");
        System.out.println(inches + " inch pizza");
        System.out.println(crust + " crust");
        System.out.println(toppings);

        //apply discount if user is eligible
        //ADD LINES FOR TASK #4 HERE

        //EDIT PROGRAM FOR TASK #5
        //SO ALL MONEY OUTPUT APPEARS WITH 2 DECIMAL PLACES
        System.out.println("The cost of your order is: $" + cost);

        //calculate and display tax and total cost
        tax = cost * TAX_RATE;
        System.out.println("The tax is:  $" + tax);
        System.out.println("The total due is:  $" + (tax+cost));

        System.out.println(
            "Your order will be ready for pickup in 30 minutes.");
    }
}
```

Chapter 4 Lab

Loops and Files

Objectives

- Be able to convert an algorithm using control structures into Java
- Be able to write a while loop
- Be able to write an do-while loop
- Be able to write a for loop
- Be able to use the Random class to generate random numbers
- Be able to use file streams for I/O
- Be able to write a loop that reads until end of file
- Be able to implement an accumulator and a counter

Introduction

This is a simulation of rolling dice. Actual results approach theory only when the sample size is large. So we will need to repeat rolling the dice a large number of times (we will use 10,000). The theoretical probability of rolling doubles of a specific number is 1 out of 36 or approximately 278 out of 10,000 times that you roll the pair of dice. Since this is a simulation, the numbers will vary a little each time you run it.

Check out how to use the random number generator (introduced in section 4.11 of the text) to get a number between 1 and 6 to create the simulation.

We will continue to use control structures that we have already learned, while exploring control structures used for repetition. We shall also continue our work with algorithms, translating a given algorithm to java in order to complete our program. We will start with a while loop, then use the same program, changing the while loop to a do-while loop, and then a for loop.

We will be introduced to file input and output. We will read a file, line by line, converting each line into a number. We will then use the numbers to calculate the mean and standard deviation.

First we will learn how to use file output to get results printed to a file. Next we will use file input to read the numbers from a file and calculate the mean. Finally, we will see that when the file is closed, and then reopened, we will start reading from the top of the file again so that we can calculate the standard deviation.

Task #1 While loop

1. Copy the file *DiceSimulation.java* (see code listing 4.1) from www.aw.com/cssupport or as directed by your instructor. *DiceSimulation.java* is incomplete. Since there is a large part of the program missing, the output will be incorrect if you run *DiceSimulation.java*.
2. I have declared all the variables. You need to add code to simulate rolling the dice and keeping track of the doubles. Convert the algorithm below to Java and place it in the main method after the variable declarations, but before the output statements. You will be using several control structures: a **while** loop and an if-else-if statement nested inside another if statement. Use the indenting of the algorithm to help you decide what is included in the loop, what is included in the if statement, and what is included in the nested if-else-if statement.
3. To "roll" the dice, use the nextInt method of the random number generator to generate an integer from 1 to 6.

```
Repeat while the number of dice rolls are less than the number of times the dice should be rolled.
        Get the value of the first die by "rolling" the first die
        Get the value of the second die by "rolling" the second die
        If the value of the first die is the same as the value of the second die
            If value of first die is 1
                Increment the number of times snake eyes were rolled
            Else if value of the first die is 2
                Increment the number of times twos were rolled
            Else if value of the first die is 3
                Increment the number of times threes were rolled
            Else if value of the first die is 4
            Increment the number of times fours were rolled
            Else if value of the first die is 5
                Increment the number of times fives were rolled
            Else if value of the first die is 6
                Increment the number of times sixes were rolled
        Increment the number of times the dice were rolled
```

4. Compile and run. You should get numbers that are somewhat close to 278 for each of the different pairs of doubles. Run it several times. You should get different results than the first time, but again it should be somewhat close to 278.

Task #2 Using Other Types of Loops

1. Change the while loop to a **do-while** loop. Compile and run. You should get the same results.
2. Change the do loop to a **for** loop. Compile and run. You should get the same results.

Task #3 Writing Output to a File

1. Copy the files *StatsDemo.java* (see code listing 4.2) and *Numbers.txt* from www.aw.com/cssupport or as directed by your instructor.
2. First we will write output to a file:
   a) Create a FileWriter object passing it the filename "Results.txt" (Don't forget the needed import statement).
   b) Create a PrintWriter object passing it the FileWriter object.
   c) Since you are using a FileWriter object, add a throws clause to the main method header.
   d) Print the mean and standard deviation to the output file using a three decimal format, labeling each.
   e) Close the output file.
3. Compile, debug, and run. You will need to type in the filename **Numbers.txt**. You should get no output to the console, but running the program will create a file called **Results.txt** with your output. The output you should get at this point is: mean = 0.000, standard deviation = 0.000. This is not the correct mean or standard deviation for the data, but we will fix this in the next tasks.

Task #4 Calculating the Mean

1. Now we need to add lines to allow us to read from the input file and calculate the mean.
   a) Create a FileReader object passing it the filename.
   b) Create a BufferedReader object passing it the FileReader object.
2. Write a priming read to read the first line of the file.
3. Write a loop that continues until you are at the end of the file.
4. The body of the loop will
   a) convert the line into a double value and add the value to the accumulator
   b) increment the counter
   c) read a new line from the file
5. When the program exits the loop close the input file.
6. Calculate and store the mean. The mean is calculated by dividing the accumulator by the counter.
7. Compile, debug, and run. You should now get a mean of 77.444, but the standard deviation will still be 0.000.

Task #5 Calculating the Standard Deviation

1. We need to reconnect to the file so that we can start reading from the top again.
   a) Create a FileReader object passing it the filename.
   b) Create a BufferedReader object passing it the FileReader object.
2. Reinitialize sum and count to 0.
3. Write a priming read to read the first line of the file.
4. Write a loop that continues until you are at the end of the file.
5. The body of the loop will
   a) convert the line into a double value and subtract the mean, store the result in difference
   b) add the square of the difference to the accumulator
   c) increment the counter
   d) read a new line from the file
6. When the program exits the loop close the input file.
7. The variance is calculated by dividing the accumulator (sum of the squares of the difference) by the counter. Calculate the standard deviation by taking the square root of the variance (Use Math.sqrt ( ) to take the square root).
8. Compile, debug, and run. You should get a mean of 77.444 and standard deviation of 10.021.

Code Listing 4.1 (DiceSimulation.java)

```
/**
   This class simulates rolling a pair of dice 10,000 times and
   counts the number of times doubles of are rolled for each differ-
   ent pair of doubles.
*/

import java.util.Random;      //to use the random number
                              //generator
public class DiceSimulation
{
    public static void main(String[] args)
    {
         final int NUMBER = 10000;   //the number of times to
                                     //roll the dice

         //a random number generator used in simulating
         //rolling a dice
         Random generator = new Random();

         int die1Value;        // number of spots on the first
                               // die
         int die2Value;        // number of spots on the second
                               // die
         int count = 0;        // number of times the dice were
                               // rolled
         int snakeEyes = 0;    // number of times snake eyes is
                               // rolled
         int twos = 0;         // number of times double
                               // two is rolled
         int threes = 0;       // number of times double three
                               // is rolled
         int fours = 0;        // number of times double four
                               // is rolled
         int fives = 0;        // number of times double five
                               // is rolled
         int sixes = 0;        // number of times double six is
                               // rolled
```

Code Listing 4.1 continued on next page.

```
          //ENTER YOUR CODE FOR THE ALGORITHM HERE

          System.out.println ("You rolled snake eyes " +
               snakeEyes + " out of " + count + " rolls.");
          System.out.println ("You rolled double twos " + twos +
               " out of " + count + " rolls.");
          System.out.println ("You rolled double threes " +
               threes + " out of " + count + " rolls.");
          System.out.println ("You rolled double fours " + fours
               + " out of " + count + " rolls.");
          System.out.println ("You rolled double fives " + fives
               + " out of " + count + " rolls.");
          System.out.println ("You rolled double sixes " + sixes
               + " out of " + count + " rolls.");
     }
}
```

Code Listing 4.2 (StatsDemo.java)

```
import java.text.DecimalFormat; //for number formatting
import java.util.Scanner;       //for keyboard input
//ADD AN IMPORT STATEMENT HERE  //for using files

public class StatsDemo
{
    public static void main(String [] args)    //ADD A THROWS
                                               //CLAUSE HERE
    {
        double sum = 0;        //the sum of the numbers
        int count = 0;         //the number of numbers added
        double mean = 0;       //the average of the numbers
        double stdDev = 0;     //the standard deviation of the
                               //numbers
        String line;           //a line from the file
        double difference;     //difference between the value
                               //and the mean

        //create an object of type Decimal Format
        DecimalFormat threeDecimals =
            new DecimalFormat("0.000");
        //create an object of type Scanner
        Scanner keyboard = new Scanner (System.in);
        String filename;   // the user input file name

        //Prompt the user and read in the file name
        System.out.println(
            "This program calculates statistics"
            + "on a file containing a series of numbers");
        System.out.print("Enter the file name:  ");
        filename = keyboard.nextLine();

        //ADD LINES FOR TASK #4 HERE
        //Create a FileReader object passing it the filename
        //Create a BufferedReader object passing it the
        //FileReader object.
```

Code Listing 4.2 continued on next page.

```
        //priming read to read the first line of the file
        //create a loop that continues until you are at the
        //end of the file
        //convert the line to double value, add the value to
        //the sum
        //increment the counter
        //read a new line from the file
        //close the input file
        //store the calculated mean

        //ADD LINES FOR TASK #5 HERE
        //create a FileReader object passing it the filename
        //create a BufferedReader object passing it the
        //FileReader object.
        //reinitialize the sum of the numbers
        //reinitialize the number of numbers added
        //priming read to read the first line of the file
        //loop that continues until you are at the end of the
        //file
        //convert the line into a double value and subtract
        //the mean
        //add the square of the difference to the sum
        //increment the counter
        //read a new line from the file
        //close the input file
        //store the calculated standard deviation


        //ADD LINES FOR TASK #3 HERE
        //create an object of type FileWriter using
        //"Results.txt"
        //create an object of PrintWriter passing it the
        //FileWriter object.
        //print the results to the output file
        //close the output file
    }
}
```

Chapter 5 Lab
Methods

Objectives

Be able to write methods
Be able to call methods
Be able to write javadoc comments
Be able to create HTML documentation for our Java class using javadoc

Introduction

Methods are commonly used to break a problem down into small manageable pieces. A large task can be broken down into smaller tasks (methods) that contain the details of how to complete that small task. The larger problem is then solved by implementing the smaller tasks (calling the methods) in the correct order.

This also allows for efficiencies, since the method can be called as many times as needed without rewriting the code each time.

Finally, we will use documentation comments for each method, and generate HTML documents similar to the Java APIs that we have seen.

Task #1 `void` Methods

1. Copy the file *Geometry.java* (code listing 5.1) from www.aw.com/cssupport or as directed by your instructor. This program will compile, but when you run it, it doesn't appear to do anything except wait. That is because it is waiting for user input, but the user doesn't have the menu to choose from yet. We will need to create this.
2. Above the main method, but in the Geometry class, create a static method called printMenu that has no parameter list and does not return a value. It will simply print out instructions for the user with a menu of options for the user to choose from. The menu should appear to the user as:

```
This is a geometry calculator
Choose what you would like to calculate
1. Find the area of a circle
2. Find the area of a rectangle
3. Find the area of a triangle
4. Find the circumference of a circle
5. Find the perimeter of a rectangle
6. Find the perimeter of a triangle
Enter the number of your choice:
```

3. Add a line in the main method that calls the printMenu method as indicated by the comments.
4. Compile, debug, and run. You should be able to choose any option, but you will always get 0 for the answer. We will fix this in the next task.

Task #2 Value-Returning Methods

1. Write a static method called **circleArea** that takes in the radius of the circle and returns the area using the formula $A = \pi r^2$.
2. Write a static method called **rectangleArea** that takes in the length and width of the rectangle and returns the area using the formula $A = lw$.
3. Write a static method called **triangleArea** that takes in the base and height of the triangle and returns the area using the formula $A = {}^1\!/_2 bh$.
4. Write a static method called **circleCircumference** that takes in the radius of the circle and returns the circumference using the formula $C = 2\pi r$.
5. Write a static method called **rectanglePerimeter** that takes in the length and the width of the rectangle and returns the perimeter of the rectangle using the formula $P = 2l + 2w$.
6. Write a static method called **trianglePerimeter** that takes in the lengths of the three sides of the triangle and returns the perimeter of the triangle which is calculated by adding up the three sides.

Task #3 Calling Methods

1. Add lines in the main method in the GeometryDemo class which will call these methods. The comments indicate where to place the method calls.
2. **Below, write some sample data and hand calculated results for you to test all 6 menu items.**
3. Compile, debug, and run. Test out the program using your sample data.

Task #4 Java Documentation

1. Write javadoc comments for each of the 7 static methods that you just wrote. They should include
   a) A one line summary of what the method does.
   a) A description of what the program requires to operate and what the result of that operation is.
   a) @param listing and describing each of the parameters in the parameter list (if any).
   a) @return describing the information that is returned to the calling statement (if any).
2. Generate the documentation. Check the method summary and the method details to ensure your comments were put into the Java Documentation correctly.

Code Listing 5.1 (Geometry.java)

```
import java.util.Scanner;

/**
   This program demonstrates static methods
*/

public class Geometry
{
   public static void main (String [] args)
   {
      int choice;          //the user's choice
      double value = 0;    //the value returned from the method
      char letter;         //the Y or N from the user's decision
                           //to exit
      double radius;       //the radius of the circle
      double length;       //the length of the rectangle
      double width;        //the width of the rectangle
      double height;       //the height of the triangle
      double base;         //the base of the triangle
      double side1;        //the first side of the triangle
      double side2;        //the second side of the triangle
      double side3;        //the third side of the triangle

      //create a scanner object to read from the keyboard
      Scanner keyboard = new Scanner (System.in);

      //do loop was chose to allow the menu to be displayed
      //first
      do
      {
         //call the printMenu method
         choice = keyboard.nextInt();

         switch (choice)
         {
```

Code Listing 5.1 continued on next page.

```
case 1:
  System.out.print(
       "Enter the radius of the circle: ");
  radius = keyboard.nextDouble();
  //call the circleArea method and
  //store the result
  //in the value
  System.out.println(
       "The area of the circle is " + value);
  break;
case 2:
  System.out.print(
       "Enter the length of the rectangle: ");
  length = keyboard.nextDouble();
  System.out.print(
       "Enter the width of the rectangle: ");
  width = keyboard.nextDouble();
  //call the rectangleArea method and store
  //the result in the value
  System.out.println(
       "The area of the rectangle is " + value);
  break;
case 3:
  System.out.print(
       "Enter the height of the triangle: ");
  height = keyboard.nextDouble();
  System.out.print(
       "Enter the base of the triangle: ");
  base = keyboard.nextDouble();
  //call the triangleArea method and store
  //the result in the value
  System.out.println(
       "The area of the triangle is " + value);
  break;
```

Code Listing 5.1 continued on next page.

```
            case 4:
              System.out.print(
                   "Enter the radius of the circle: ");
              radius = keyboard.nextDouble();
              //call the circumference method and
              //store the result in the value
              System.out.println(
                   "The circumference of the circle is " + value);
              break;
            case 5:
              System.out.print(
                   "Enter the length of the rectangle: ");
              length = keyboard.nextDouble();
              System.out.print(
                   "Enter the width of the rectangle: ");
              width = keyboard.nextDouble();
              //call the perimeter method and store the result
              //in the value
              System.out.println(
                   "The perimeter of the rectangle is " + value);
              break;
            case 6:
              System.out.print("Enter the length of side 1 " +
                   "of the triangle:  ");
              side1 = keyboard.nextDouble();
              System.out.print("Enter the length of side 2 " +
                   "of the triangle:  ");
              side2 = keyboard.nextDouble();
              System.out.print("Enter the length of side 3 " +
                   "of the triangle:  ");
              side3 = keyboard.nextDouble();
              //call the perimeter method and store the result
              //in the value
              System.out.println("The perimeter of the " +
                   "triangle is " + value);
              break;
            default:
```

Code Listing 5.1 continued on next page.

```
               System.out.println(
                     "You did not enter a valid choice.");
          }

          //consumes the new line character after the number
          keyboard.nextLine();

          System.out.println("Do you want to exit the program " +
              "(Y/N)?:  ");
          String answer = keyboard.nextLine();
          letter = answer.charAt(0);
      }while (letter != ‘Y’ && letter != ‘y’);
   }
}
```

Chapter 6 Lab
Classes and Objects

Objectives

- Be able to declare a new class
- Be able to write a constructor
- Be able to write instance methods that return a value
- Be able to write instance methods that take arguments
- Be able to instantiate an object
- Be able to use calls to instance methods to access and change the state of an object

Introduction

Everyone is familiar with a television. It is the object we are going to create in this lab. First we need a blueprint. All manufacturers have the same basic elements in the televisions they produce as well as many options. We are going to work with a few basic elements that are common to all televisions. Think about a television in general. It has a brand name (i.e. it is made by a specific manufacturer). The television screen has a specific size. It has some basic controls. There is a control to turn the power on and off. There is a control to change the channel. There is also a control for the volume. At any point in time, the television's state can be described by how these controls are set.

We will write the television class. Each object that is created from the television class must be able to hold information about that instance of a television in fields. So a television object will have the following attributes:

- **manufacturer**. The `manufacturer` attribute will hold the brand name. This cannot change once the television is created, so will be a named constant.
- **screenSize.** The `screenSize` attribute will hold the size of the television screen. This cannot change once the television has been created so will be a named constant.
- **powerOn.** The `powerOn` attribute will hold the value true if the power is on, and false if the power is off.
- **channel.** The `channel` attribute will hold the value of the station that the television is showing.
- **volume.** The `volume` attribute will hold a number value representing the loudness (0 being no sound).

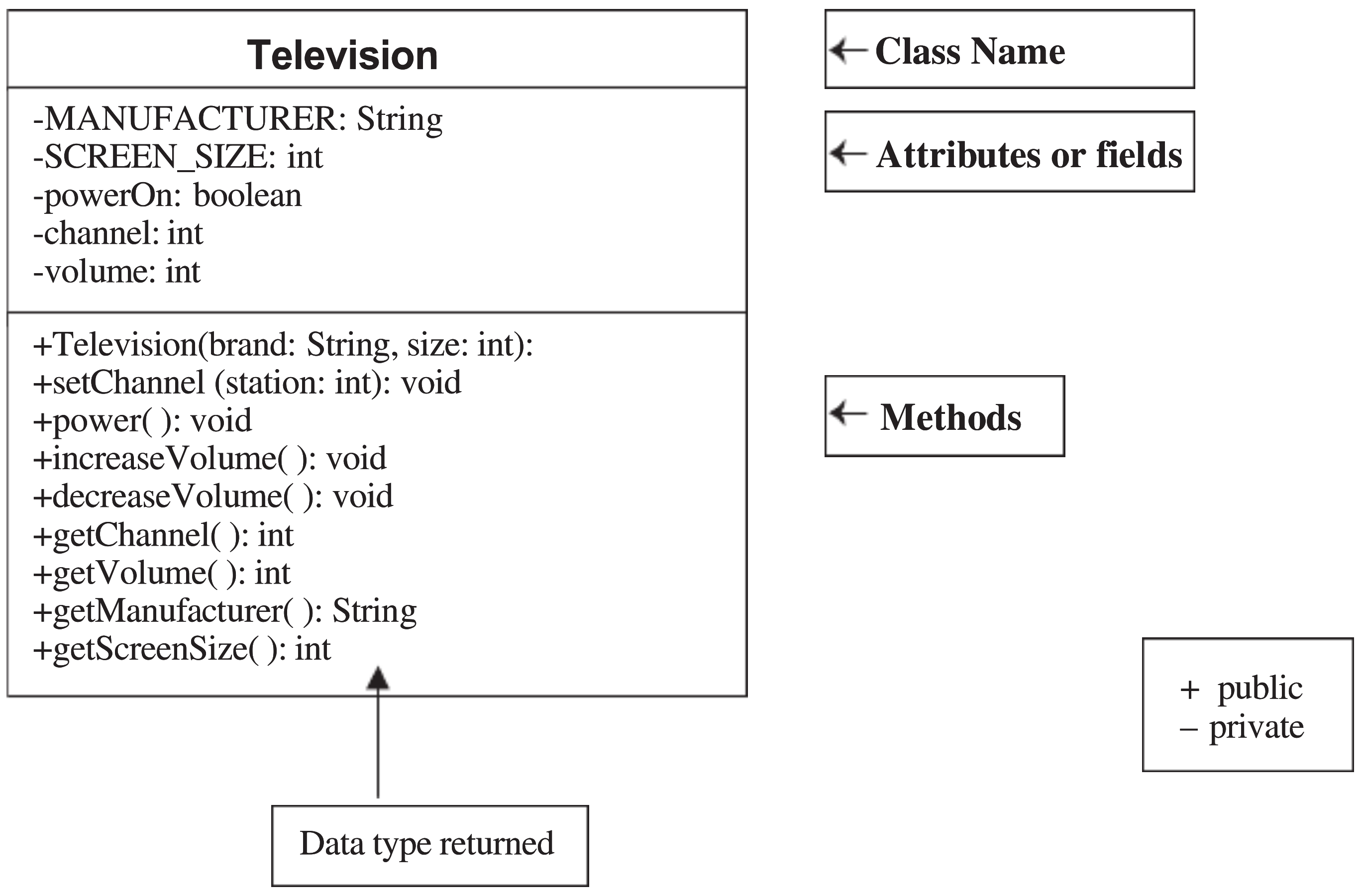
These attributes become **fields** in our class.

The television object will also be able to control the state of its attributes. These controls become **methods** in our class.

- **setChannel.** The `setChannel` method will store the desired station in the channel field.
- **power.** The `power` method will toggle the power between on and off, changing the value stored in the `powerOn` field from true to false or from false to true.
- **increaseVolume.** The `increaseVolume` method will increase the value stored in the `volume` field by 1.
- **decreaseVolume.** The `decreaseVolume` method will decrease the value stored in the `volume` field by 1.
- **getChannel.** The `getChannel` method will return the value stored in the channel field.
- **getVolume.** The `getVolume` method will return the value stored in the volume field.
- **getManufacturer.** The `getManufacturer` method will return the constant value stored in the `MANUFACTURER` field.
- **getScreenSize.** The `getScreenSize` method will return the constant value stored in the `SCREEN_SIZE` field.

We will also need a constructor method that will be used to create an instance of a Television.

These ideas can be brought together to form a UML (Unified Modeling Language) diagram for this class as shown below.



Task #1 Creating a New Class

1. In a new file, create a class definition called Television.
2. Put a program header (comments/documentation) at the top of the file

   ```
   // The purpose of this class is to model a television
   // Your name and today's date
   ```

3. Declare the 2 constant fields listed in the UML diagram.
4. Declare the 3 remaining fields listed in the UML diagram.
5. Write a comment for each field indicating what it represents.
6. Save this file as *Television.java*.
7. Compile and debug. Do not run.

Task #2 Writing a Constructor

1. Create a constructor definition that has two parameters, a manufacturer's brand and a screen size. These parameters will bring in information
2. Inside the constructor, assign the values taken in from the parameters to the corresponding fields.
3. Initialize the `powerOn` field to false (power is off), the `volume` to 20, and the `channel` to 2.
4. Write comments describing the purpose of the constructor above the method header.
5. Compile and debug. Do not run.

Task #3 Methods

1. Define accessor methods called `getVolume`, `getChannel`, `getManufacturer`, and `getScreenSize` that return the value of the corresponding field.
2. Define a mutator method called `setChannel` accepts a value to be stored in the channel field.
3. Define a mutator method called power that changes the state from true to false or from false to true. This can be accomplished by using the `NOT` operator (!). If the `boolean` variable `powerOn` is true, then `!powerOn` is false and vice versa. Use the assignment statement

   ```
   powerOn = !powerOn;
   ```

   to change the state of `powerOn` and then store it back into `powerOn` (remember assignment statements evaluate the right hand side first, then assign the result to the left hand side variable.
4. Define two mutator methods to change the volume. One method should be called `increaseVolume` and will increase the volume by 1. The other method should be called `decreaseVolume` and will decrease the volume by 1.
5. Write javadoc comments above each method header.
6. Compile and debug. Do not run.

Task #4 Running the application

1. You can only execute (run) a program that has a main method, so there is a driver program that is already written to test out your `Television` class. Copy the file *TelevisionDemo.java* (see code listing 3.1) from www.aw.com/cssupport or as directed by your instructor. Make sure it is in the same directory as *Television.java*.
2. Compile and run TelevisionDemo and follow the prompts.
3. If your output matches the output below, *Television.java* is complete and correct. You will not need to modify it further for this lab.

OUTPUT (boldface is user input)

```
A 55 inch Toshiba has been turned on.
What channel do you want? 56
Channel: 56  Volume: 21
Too loud!! I am lowering the volume.
Channel: 56  Volume: 15
```

Task #5 Creating another instance of a Television

1. Edit the *TelevisionDemo.java* file.
2. Declare another Television object called portable.
3. Instantiate portable to be a Sharp 19 inch television.
4. Use a call to the power method to turn the power on.
5. Use calls to the `accessor` methods to print what television was turned on.
6. Use calls to the `mutator` methods to change the channel to the user's preference and decrease the volume by two.
7. Use calls to the `accessor` methods to print the changed state of the portable.
8. Compile and debug this class.
9. Run TelevisionDemo again.
10. The output for task #5 will appear after the output from above, since we added onto the bottom of the program. The output for task #5 is shown below.

OUTPUT (boldface is user input)

```
A 19 inch Sharp has been turned on.
What channel do you want? 7
Channel: 7  Volume: 18
```

Code Listing 6.1 (TelevisionDemo.java)

```
/** This class demonstrates the Television class*/

import java.util.Scanner;

public class TelevisionDemo
{
    public static void main(String[] args)
    {
         //create a Scanner object to read from the keyboard
         Scanner keyboard = new Scanner (System.in);

         //declare variables
         int station;   //the user’s channel choice

         //declare and instantiate a television object
         Television bigScreen = new Television("Toshiba", 55);
         //turn the power on
         bigScreen.power();
         //display the state of the television
         System.out.println("A " + bigScreen.getScreenSize() +
             bigScreen.getManufacturer()   +
             " has been turned on.");
         //prompt the user for input and store into station
         System.out.print("What channel do you want?  ");
         station = keyboard.nextInt();

         //change the channel on the television
         bigScreen.setChannel(station);
         //increase the volume of the television
         bigScreen.increaseVolume();
         //display the the current channel and volume of the
         //television
         System.out.println("Channel:  " +
             bigScreen.getChannel() +
             "   Volume:  "   + bigScreen.getVolume());
```

Code Listing 6.1 continued on next page.

```
            System.out.println(
                "Too loud!! I am lowering the volume.");
            //decrease the volume of the television
            bigScreen.decreaseVolume();
            bigScreen.decreaseVolume();
            bigScreen.decreaseVolume();
            bigScreen.decreaseVolume();
            bigScreen.decreaseVolume();
            bigScreen.decreaseVolume();
            //display the current channel and volume of the
            //television
            System.out.println("Channel:  " +
                bigScreen.getChannel() +
                "   Volume:  "   + bigScreen.getVolume());
            System.out.println();  //for a blank line

            //HERE IS WHERE YOU DO TASK #5
        }
    }
```

Chapter 7 Lab
GUI Applications

Objectives

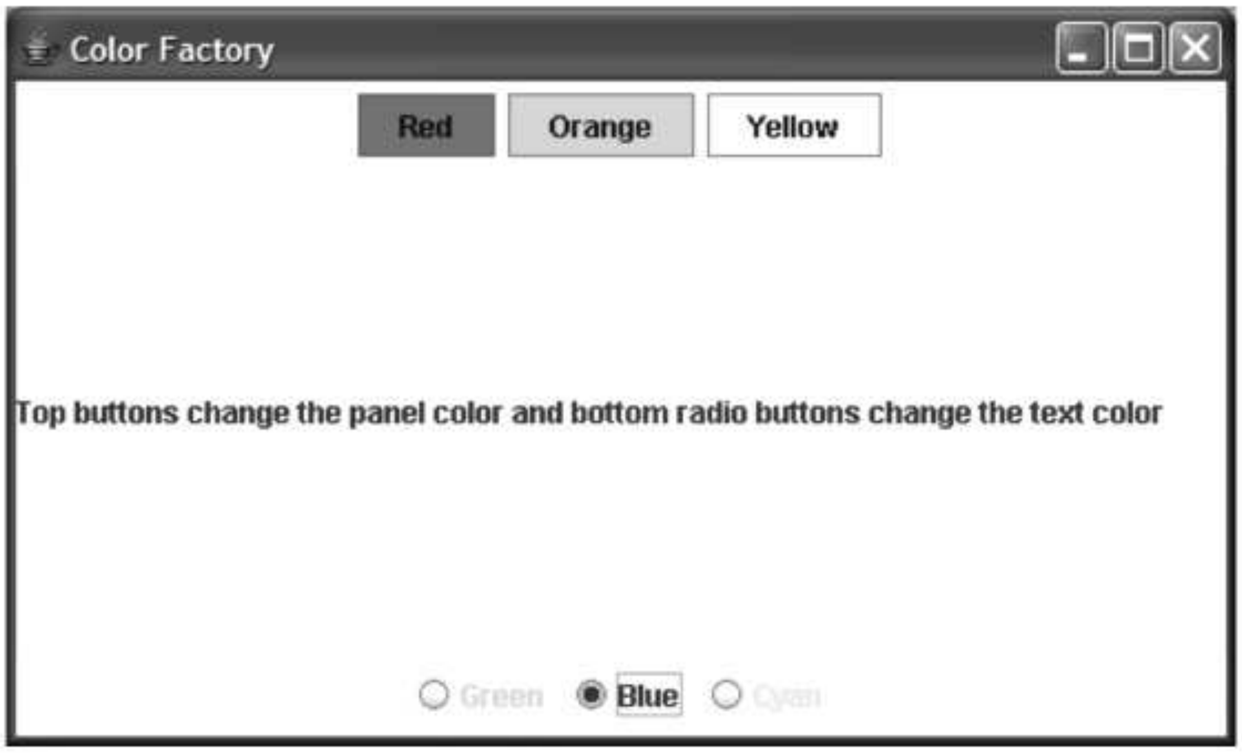
- Be able to create a closeable window
- Be able to create panels containing buttons
- Be able to use different layouts
- Be able to handle button events

Introduction

In this lab, we will be creating a graphical user interface (GUI) to allow the user to select a button that will change the color of the center panel and radio buttons that will change the color of the text in the center panel. We will need to use a variety of Swing components to accomplish this task.

We will build two panels, a top panel containing three buttons and a bottom panel containing three radio buttons. Layouts will be used to add these panels to the window in the desired positions. A label with instructions will also be added to the window. Listeners will be employed to handle the events desired by the user.

Our final GUI should look like the following



Top buttons change the panel color and bottom radio buttons change the text color

Task #1 Creating a GUI

1. Import the required Java libraries.
2. Create a class called ColorFactory that inherits from JFrame.
3. Create named constants for a width of 500 and height of 300 for the frame.
4. Write a default constructor that does the following
   a) Set the title of the window to Color Factory.
   b) Set the size of the window using the constants.
   c) Specify what happens when the close button is clicked.
   d) Get the content pane of the JFrame and set the layout manager to border layout.
   e) Call the method to build the top panel (to be written as directed below).
   f) Add the panel to the north part of the content pane.
   g) Call the method to build the bottom panel (to be written as directed below).
   h) Add this panel to the south part of the content pane.
   i) Create a label that contains the message “Top buttons change the panel color and bottom radio buttons change the text color.”
   j) Add this label to the center part of the content pane.

Task #2 Writing Private Methods

1. Write a private method that builds the top panel as follows
   a) Create a panel that contains three buttons, red, orange, and yellow.
   b) Use flow layout for the panel and set the background to be white.
   c) The buttons should be labeled with the color name and also appear in that color.
   d) Set the action command of each button to be the first letter of the color name.
   e) Add button listener that implements action listener for each button.
2. Create a bottom panel in the same way as the top panel above, but use radio buttons with the colors green, blue, and cyan.

Task #3 Writing Inner Classes

1. Write a private inner class called ButtonListener that implements ActionListener. It should contain an actionPerformed method to handle the button events. This event handler will handle all button events, so you must get the action command of the event and write a decision structure to determine which color to set the background of the content pane.
2. Write another private inner class called RadioButtonListener, similar to Button listener. It will handle all radio button events, so you will need to check the source of the event and write a decision structure to determine which color should be used for the text of the message.

Task #4 Running the GUI Program

1. Write a main method that declares and creates one instance of a ColorFactory, then use the setVisible method to show it on the screen.

Chapter 8 Lab
Arrays

Objectives

- Be able to declare and instantiate arrays
- Be able to fill an array using a for loop
- Be able to access and process data in an array
- Be able to write a sorting method
- Be able to use an array of objects

Introduction

Everyone is familiar with a list. We make shopping lists, to-do lists, assignment lists, birthday lists, etc. Notice that though there may be many items on the list, we call the list by one name. That is the idea of the array, one name for a list of related items. In this lab, we will work with lists in the form of an array.

It will start out simple with a list of numbers. We will learn how to process the contents of an array. We will also explore sorting algorithms, using the selection sort. We will then move onto more complicated arrays, arrays that contain objects.

Task #1 Average Class

Create a class called Average according to the UML diagram.

Average
-data [ ] :int -mean: double
+Average( ): +calculateMean( ): void +toString( ): String +selectionSort( ): void

This class will allow a user to enter 5 scores into an array. It will then rearrange the data in descending order and calculate the mean for the data set.

Attributes:
- **`data[]`**—the array which will contain the scores
- **`mean`**—the arithmetic average of the scores

Methods:
- **`Average`**—the constructor. It will allocate memory for the array. Use a for loop to repeatedly display a prompt for the user which should indicate that user should enter score number 1, score number 2, etc. Note: The computer starts counting with 0, but people start counting with 1, and your prompt should account for this. For example, when the user enters score number 1, it will be stored in indexed variable 0. The constructor will then call the **`selectionSort`** and the **`calculateMean`** methods.
- **`calculateMean`**—this is a method that uses a for loop to access each score in the array and add it to a running total. The total divided by the number of scores (use the length of the array), and the result is stored into mean.
- **`toString`**—returns a String containing data in descending order and the mean.
- **`selectionSort`**—this method uses the selection sort algorithm to rearrange the data set from highest to lowest.

Task #2 Average Driver

1. Create an **`AverageDriver`** class. This class only contains the main method. The main method should declare and instantiate an Average object. The Average object information should then be printed to the console.
2. Compile, debug, and run the program. It should output the data set from highest to lowest and the mean. Compare the computer's output to your hand calculation using a calculator. If they are not the same, do not continue until you correct your code.

Task #3 Arrays of Objects

1. Copy the files *Song.java* (code listing 8.1), *CompactDisc.java* (code listing 8.2) and *Classics.txt* (code listing 8.3) from www.aw.com/cssupport or as directed by your instructor. *Song.java* is complete and will not be edited. *Classics.txt* is the data file that will be used by *CompactDisc.java*, the file you will be editing.
2. In *CompactDisc.java*, there are comments indicating where the missing code is to be placed. Declare an array of Songs, called cd, to be of size 6.
3. Fill the array by creating a new song with the title and artist and storing it in the appropriate position in the array.
4. Print the contents of the array to the console.
5. Compile, debug, and run. Your output should be as follows:

```
Contents of Classics
Ode to Joy by Bach
The Sleeping Beauty by Tchaikovsky
Lullaby by Brahms
Canon by Bach
Symphony No. 5 by Beethoven
The Blue Danube Waltz by Strauss
```

Code Listing 8.1 (Song.java)

```
/*This program represents a song*/
public class Song
{
    /**The title of the song*/
    private String title;
    /**The artist who sings the song*/
    private String artist;

    /**constructor
    @param title The title of the song
    @param artist The artost who sings the song*/
    public Song(String title, String artist)
    {
        this.title = title;
        this.artist = artist;
    }

    /**toString method returns a description of the song
    @return a String containing the name of the song and the artist*/
    public String toString()
    {
        return title + " by " + artist + "\n";
    }
}
```

Code Listing 8.2 (CompactDisc.java)

```
/*This program creates a list of songs for a CD by reading from a
file*/
import java.io.*;

public class CompactDisc
{
    public static void main(String [] args) throws IOException
    {
        FileReader file = new FileReader("Classics.txt");
        BufferedReader input = new BufferedReader(file);
        String title;
        String artist;

        //Declare an array of songs, called cd, of size 6

        for (int i = 0; i  < cd.length; i++)
        {
            title = input.readLine();
            artist = input.readLine();
            // fill the array by creating a new song with
            // the title and artist and storing it in the
            // appropriate position in the array
        }

        System.out.println("Contents of Classics:");
        for (int i = 0; i < cd.length; i++)
        {
            //print the contents of the array to the console
        }
    }
}
```

Code Listing 8.3 (Classics.txt)

```
Ode to Joy
Bach
The Sleeping Beauty
Tchaikovsky
Lullaby
Brahms
Canon
Bach
Symphony No. 5
Beethoven
The Blue Danube Waltz
Strauss
```

Chapter 9 Lab
More Classes and Objects

Objectives

- Be able to write a copy constructor
- Be able to write `equals` and `toString` methods
- Be able to use objects made up of other objects (Aggregation)
- Be able to write methods that pass and return objects

Introduction

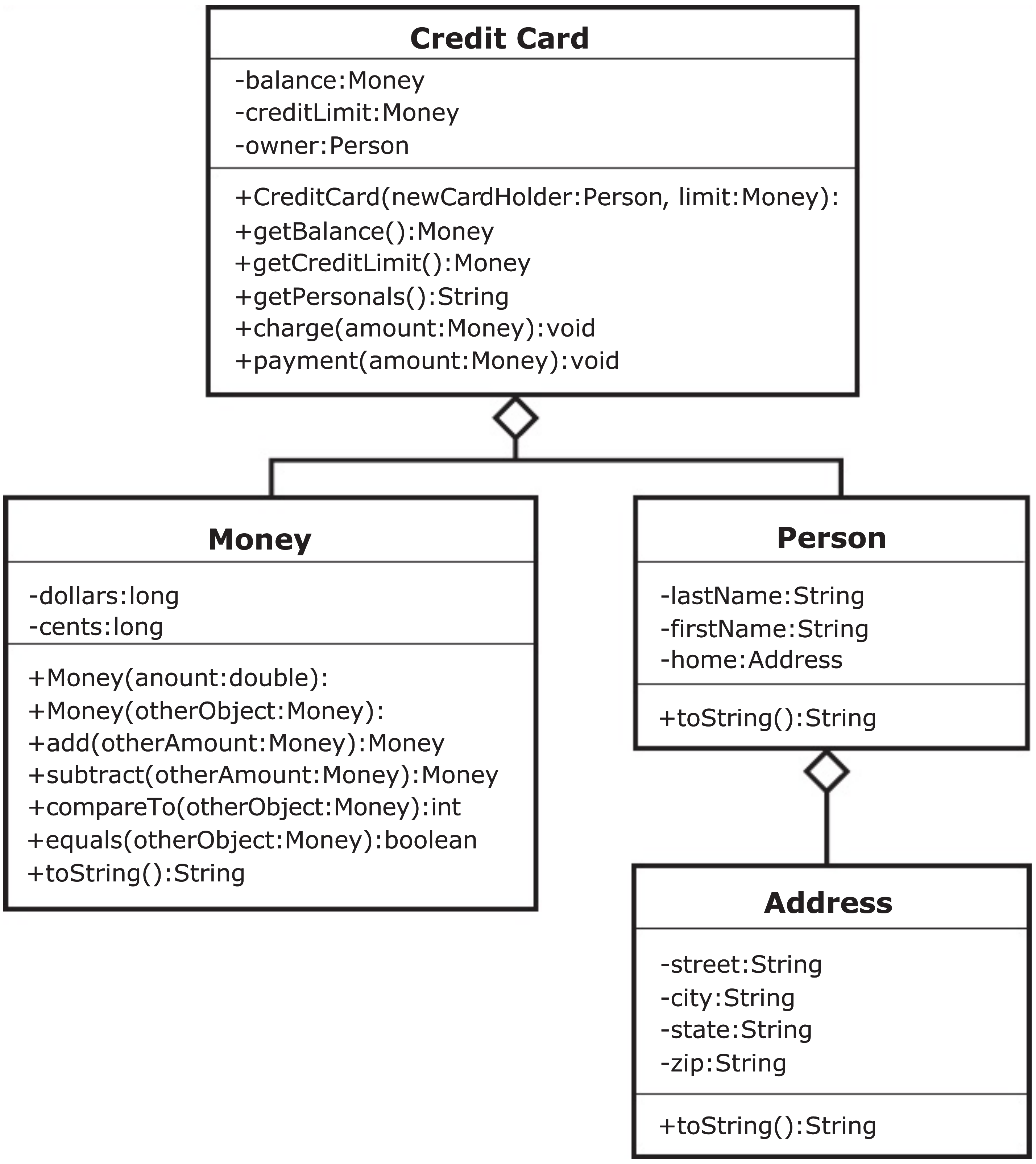
We discussed objects in Chapter 6 and we modeled a television in the Chapter 6 lab. We want build on that lab, and work more with objects. This time, the object that we are choosing is more complicated. It is made up of other objects. This is called aggregation. A credit card is an object that is very common, but not as simple as a television. Attributes of the credit card include information about the owner, as well as a balance and credit limit. These things would be our instance fields. A credit card allows you to make payments and charges. These would be methods. As we have seen before, there would also be other methods associated with this object in order to construct the object and access its fields.

Examine the UML diagram that follows. Notice that the instance fields in the CreditCard class are other types of objects, a Person object or a Money object. We can say that the CreditCard "has a" Person, which means aggregation, and the Person object "has a" Address object as one of its instance fields. This aggregation structure can create a very complicated object. We will try to keep this lab reasonably simple.

To start with, we will be editing a partially written class, Money. The constructor that you will be writing is a copy constructor. This means it should create a new object, but with the same values in the instance variables as the object that is being copied.

Next, we will write `equals` and `toString` methods. These are very common methods that are needed when you write a class to model an object. You will also see a `compareTo` method that is also a common method for objects.

After we have finished the Money class, we will write a CreditCard class. This class contains Money objects, so you will use the methods that you have written to complete the Money class. The CreditCard class will explore passing objects and the possible security problems associated with it. We will use the copy constructor we wrote for the Money class to create new objects with the same information to return to the user through the accessor methods.



Task #1 Writing a Copy Constructor

1. Copy the files *Address.java* (code listing 9.1), *Person.java* (code listing 9.2), *Money.java* (code listing 9.3), *MoneyDriver.java* (code listing 9.4), and *CreditCardDemo.java* (code listing 9.5) from www.aw.com/cssupport or as directed by your instructor. *Address.java*, *Person.java*, *MoneyDemo.java*, and *CreditCardDemo.java* are complete and will not need to be modified. We will start by modifying *Money.java*.
2. Overload the constructor. The constructor that you will write will be a copy constructor. It should use the parameter money object to make a duplicate money object, by copying the value of each instance variable from the parameter object to the instance variable of the new object.

Task #2 Writing `equals` and `toString` methods

1. Write and document an `equals` method. The method compares the instance variables of the calling object with instance variables of the parameter object for equality and returns true if the dollars and the cents of the calling object are the same as the dollars and the cents of the parameter object. Otherwise, it returns false.
2. Write and document a `toString` method. This method will return a String that looks like money, including the dollar sign. Remember that if you have less than 10 cents, you will need to put a 0 before printing the cents so that it appears correctly with 2 decimal places.
3. Compile, debug, and test by running the *MoneyDriver.java* driver program. You should get the output:

```
The current amount is $500.00
Adding $10.02 gives $510.02
Subtracting $10.88 gives $499.14
$10.02 equals $10.02
$10.88 does not equal $10.02
```

Task #3 Passing and Returning Objects

1. Create a CreditCard class according to the UML Diagram on the back. It should have data fields that include an owner of type Person, a balance of type Money, and a creditLimit of type Money.
2. It should have a constructor that has two parameters, a Person to initialize the owner and a Money value to initialize the creditLimit. The balance can be initialized to a Money value of zero. Remember you are passing in objects (pass by reference), so you have passed in the address to an object. If you want your CreditCard to have its own creditLimit and balance, you should create a new object of each using the copy constructor in the Money class.
3. It should have accessor methods to get the balance and the available credit. Since these are objects (pass by reference), we don't want to create an insecure credit card by passing out addresses to components in our credit card, so we must return a new object with the same values. Again, use the copy constructor to create a new object of type money that can be returned.
4. It should have an accessor method to get the information about the owner, but in the form of a String that can be printed out. This can be done by calling the toString method for the owner (who is a Person).
5. It should have a method that will charge to the credit card by adding the amount of Money in the parameter to the balance if it will not exceed the credit limit. If the credit limit will be exceeded, the amount should not be added, and an error message can be printed to the console.
6. It should have a method that will make a payment on the credit card by subtracting the amount of Money in the parameter from the balance.
7. Compile, debug, and test it out completely by running *CreditCardDemo.java*. You should get the output:

```
Diane Christie, 237J Harvey Hall, Menomonie, WI 54751
Balance: $0.00
Credit Limit: $1000.00
Attempt to charge $200.00
Charge: $200.00
Balance: $200.00
Attempt to charge $10.02
Charge: $10.02
Balance: $210.02
Attempt to pay $25.00
Payment: $25.00
Balance: $185.02
Attempt to charge $990.00
Exceeds credit limit
Balance: $185.02
```

Code Listing 9.1 (Address.java)

```
/**Defines an address using a street, city, state, and zipcode*/
public class Address
{
    /**The street number and street name*/
    private String street;

    /**The city in which the address is located*/
    private String city;

    /**The state in which the address is located*/
    private String state;

    /**The zip code associated with that city and street*/
    private String zip;

    /**Constructor creates an address using four parameters
    @param road describes the street number and name
    @param town describes the city
    @param st describes the state
    @param zipCode describes the zip code*/
    public Address(String road, String town, String st,
         String zipCode)
    {
         street = road;
         city = town;
         state = st;
         zip = zipCode;
    }

    /**toString method returns information about the address
    @return all imformation about the address*/
    public String toString()
    {
         return (street + ", " + city + ", " + state + " " +
              zip);
    }
}
```

Code Listing 9.2 (Person.java)

```
/**Defines a person by name and address*/
public class Person
{
    /**The person’s last name*/
    private String lastName;

    /**The person’s first name*/
    private String firstName;

    /**The person’s address*/
    private Address home;

    /**Constructor creates a person from a last name,
    first name, and address
    @param last the person’s last name
    @param first the person’s first name
    @param residence the person’s address*/
    public Person(String last, String first, Address residence)
    {
        lastName = last;
        firstName = first;
        home = residence;
    }

    /**toString method returns information about the person
    @return information about the person*/
    public String toString()
    {
        return(firstName + " " + lastName + ", " +
             home.toString());
    }

}
```

Code Listing 9.3 (Money.java)

```
/**Objects represent nonnegative amounts of money*/
public class Money
{
    /**A number of dollars*/
    private long dollars;
    /**A number of cents*/
    private long cents;

    /**Constructor creates a Money object using the amount of
    money in dollars and cents represented with a decimal
    number
    @param amount the amount of money in the conventional
    decimal format*/
    public Money(double amount)
    {
        if (amount < 0)
        {
            System.out.println("Error: Negative amounts " +
                "of money are not allowed.");
            System.exit(0);
        }
        else
        {
            long allCents = Math.round(amount*100);
            dollars = allCents/100;
            cents = allCents%100;
        }
    }

    /**Adds the calling Money object to the parameter Money object.
    @param otherAmount the amount of money to add
    @return the sum of the calling Money object and the
    parameter Money object*/
    public Money add(Money otherAmount)
    {
```

Code Listing 9.3 continued on next page.

```
          Money sum = new Money(0);
          sum.cents = this.cents + otherAmount.cents;
          long carryDollars = sum.cents/100;
          sum.cents = sum.cents%100;
          sum.dollars = this.dollars
               + otherAmount.dollars + carryDollars;
          return sum;
     }

     /**Subtracts the parameter Money object from the calling
     Money object and returns the difference.
     @param amount the amount of money to subtract
     @return the difference between the calling Money object
     and the parameter Money object*/
     public Money subtract (Money amount)
     {
          Money difference = new Money(0);
          if (this.cents < amount.cents)
          {
               this.dollars = this.dollars - 1;
               this.cents = this.cents + 100;
          }
          difference.dollars = this.dollars - amount.dollars;
          difference.cents = this.cents - amount.cents;
          return difference;
     }

     /**Compares instance variable of the calling object with
     the parameter object.  It returns -1 if the dollars and the
     cents of the calling object are less than the dollars and
     the cents of the parameter object, 0 if the dollars and the
     cents of the calling object are equal to the dollars and
     cents of the parameter object, and 1 if the dollars and the
     cents of the calling object are more than the dollars and
     the cents of the parameter object.
     @param amount the amount of money to compare against
     @return -1 if the dollars and the cents of the calling
```

Code Listing 9.3 continued on next page.

```
    object are less than the dollars and the cents of the
    parameter object, 0 if the dollars and the cents of the
    calling object are equal to the dollars and cents of the
    parameter object, and 1 if the dollars and the cents of the
    calling object are more than the dollars and the cents of
    the parameter object.*/
    public int compareTo(Money amount)
    {
        int value;
        if(this.dollars < amount.dollars)
        {
            value = -1;
        }
        else if (this.dollars > amount.dollars)
        {
            value = 1;
        }
        else if (this.cents < amount.dollars)
        {
            value = -1;
        }
        else if (this.cents > amount.cents)
        {
            value = 1;
        }
        else
        {
            value = 0;
        }
        return value;
    }
}
```

Code Listing 9.4 (MoneyDriver.java)

```
/**This program tests the money class.*/
public class MoneyDriver
{
    //This is a driver for testing the class
    public static void main(String[] args)
    {
         final int BEGINNING = 500;
         final Money FIRST_AMOUNT = new Money(10.02);
         final Money SECOND_AMOUNT = new Money(10.02);
         final Money THIRD_AMOUNT = new Money(10.88);
         Money balance = new Money(BEGINNING);
         System.out.println("The current amount is " +
              balance.toString());
         balance = balance.add(SECOND_AMOUNT);
         System.out.println("Adding " + SECOND_AMOUNT +
              " gives " + balance.toString());
         balance = balance.subtract(THIRD_AMOUNT);
         System.out.println("Subtracting " + THIRD_AMOUNT +
              " gives " + balance.toString());
         boolean equal = SECOND_AMOUNT.equals(FIRST_AMOUNT);
         if(equal)
              System.out.println(SECOND_AMOUNT + " equals "
                   + FIRST_AMOUNT);
         else
              System.out.println(SECOND_AMOUNT +
                   " does not equal " + FIRST_AMOUNT);

         equal = THIRD_AMOUNT.equals(FIRST_AMOUNT);
         if(equal)
              System.out.println(THIRD_AMOUNT + " equals " +
                   FIRST_AMOUNT);
         else
              System.out.println(THIRD_AMOUNT +
                   " does not equal " + FIRST_AMOUNT);
    }
}
```

Code Listing 9.5 (CreditCardDemo.java)

```java
/**Demonstrates the CreditCard class*/
public class CreditCardDemo
{
    public static void main(String[] args)
    {
        final Money LIMIT = new Money(1000);
        final Money FIRST_AMOUNT = new Money(200);
        final Money SECOND_AMOUNT = new Money(10.02);
        final Money THIRD_AMOUNT = new Money(25);
        final Money FOURTH_AMOUNT = new Money(990);
        Person owner = new Person("Christie", "Diane",
            new Address("237J Harvey Hall", "Menomonie",
            "WI", "54751"));
        CreditCard visa = new CreditCard(owner, LIMIT);
        System.out.println(visa.getPersonals());
        System.out.println("Balance: " + visa.getBalance());
        System.out.println("Credit Limit: "
            + visa.getCreditLimit());
        System.out.println();
        System.out.println("Attempt to charge " +
            FIRST_AMOUNT);
        visa.charge(FIRST_AMOUNT);
        System.out.println("Balance: " + visa.getBalance());
        System.out.println("Attempt to charge " +
            SECOND_AMOUNT);
        visa.charge(SECOND_AMOUNT);
        System.out.println("Balance: " + visa.getBalance());
        System.out.println("Attempt to pay " + THIRD_AMOUNT);
        visa.payment(THIRD_AMOUNT);
        System.out.println("Balance: " + visa.getBalance());
        System.out.println("Attempt to charge " +
            FOURTH_AMOUNT);
        visa.charge(FOURTH_AMOUNT);
        System.out.println("Balance: " + visa.getBalance());
    }
}
```

Chapter 10 Lab
Text Processing and Wrapper Classes

Objectives

- Use methods of the Character class and String class to process text
- Be able to use the StringTokenizer and StringBuffer classes

Introduction

In this lab we ask the user to enter a time in military time (24 hours). The program will convert and display the equivalent conventional time (12 hour with AM or PM) for each entry if it is a valid military time. An error message will be printed to the console if the entry is not a valid military time.

Think about how you would convert any military time 00:00 to 23:59 into conventional time. Also think about what would be valid military times. To be a valid time, the data must have a specific form. First, it should have exactly 5 characters. Next, only digits are allowed in the first two and last two positions, and that a colon is always used in the middle position. Next, we need to ensure that we never have over 23 hours or 59 minutes. This will require us to separate the substrings containing the hours and minutes. When converting from military time to conventional time, we only have to worry about times that have hours greater than 12, and we do not need to do anything with the minutes at all. To convert, we will need to subtract 12, and put it back together with the colon and the minutes, and indicate that it is PM. Keep in mind that 00:00 in military time is 12:00 AM (midnight) and 12:00 in military time is 12:00 PM (noon).

We will need to use a variety of Character class and String class methods to validate the data and separate it in order to process it. We will also use a Character class method to allow the user to continue the program if desired.

The String Tokenizer class will allow us to process a text file in order to decode a secret message. We will use the first letter of every 5th token read in from a file to reveal the secret message.

Task #1 Character and String Class Methods

1. Copy the files *Time.java* (code listing 10.1) and *TimeDemo.java* (code listing 10.2) from www.aw.com/cssupport or as directed by your instructor.
2. In the *Time.java* file, add conditions to the decision structure which validates the data. Conditions are needed that will
   a) Check the length of the string
   b) Check the position of the colon
   c) Check that all other characters are digits
3. Add lines that will separate the string into two substrings containing hours and minutes. Convert these substrings to integers and save them into the instance variables.
4. In the TimeDemo class, add a condition to the loop that converts the user's answer to a capital letter prior to checking it.
5. Compile, debug, and run. Test out your program using the following valid input: 00:00, 12:00, 04:05, 10:15, 23:59, 00:35, and the following invalid input: 7:56, 15:78, 08:60, 24:00, 3e:33, 1:111.

Task #2 StringTokenizer and StringBuffer classes

1. Copy the file *secret.txt* (code listing 10.3) from www.aw.com/cssupport or as directed by your instructor. This file is only one line long. It contains 2 sentences.
2. Write a main method that will read the file *secret.txt*, separate it into word tokens.
3. You should process the tokens by taking the first letter of every fifth word, starting with the first word in the file. These letters should converted to capitals, then be appended to a StringBuffer object to form a word which will be printed to the console to display the secret message.

Code Listing 10.1 (Time.java)

```
/**Represents time in hours and minutes using
the customary conventions*/
public class Time
{
    /**hours in conventional time*/
    private int hours;
    /**minutes in conventional time*/
    private int minutes;
    /**true if afternoon time, false if morning time*/
    private boolean afternoon;

    /**Constructs a cutomary time (12 hours, am or pm)
    from a military time ##:##
    @param militaryTime in the military format ##:##*/
    public Time(String militaryTime)
    {
         //Check to make sure something was entered
         if (militaryTime == null)
         {
              System.out.println(
                  "You must enter a valid military time." );
         }
         //Check to make sure there are 5 characters
         else if (//CONDITION TO CHECK LENGTH OF STRING)
         {
              System.out.println(militaryTime +
                  " is not a valid military time." );
         }
         else
         {
              //Check to make sure the colon is in
              //the correct spot
              if (//CONDITION TO CHECK COLON POSITION)
              {
                  System.out.println(militaryTime +
                       " is not a valid military time." );
              }
```

Code Listing 10.1 continued on next page

```
//Check to make sure all other characters are
//digits
else if (//CONDITION TO CHECK FOR DIGIT)
{
    System.out.println(militaryTime +
         " is not a valid military time." );
}
else if (//CONDITION TO CHECK FOR DIGIT)
{
    System.out.println(militaryTime +
         " is not a valid military time." );
}
else if (//CONDITION TO CHECK FOR DIGIT)
{
    System.out.println(militaryTime +
         " is not a valid military time." );
}
else if (//CONDITION TO CHECK FOR DIGIT)
{
    System.out.println(militaryTime +
         " is not a valid military time." );
}
else
{
    //SEPARATE THE STRING INTO THE HOURS
    //AND THE MINUTES, CONVERTING THEM TO
    //INTEGERS AND STORING INTO THE
    //INSTANCE VARIABLES

    //validate hours and minutes are valid
    //values
    if(hours > 23)
    {
         System.out.println(militaryTime +
              " is not a valid military" +
              " time." );
    }
```

Code Listing 10.1 continued on next page

```
                        else if(minutes > 59)
                        {
                              System.out.println(militaryTime +
                                    " is not a valid military" +
                                    " time." );
                        }
                        //convert military time to conventional
                        //time for afternoon times
                        else if (hours > 12)
                        {
                              hours = hours - 12;
                              afternoon = true;
                              System.out.println(this.toString());
                        }
                        //account for midnight
                        else if (hours == 0)
                        {
                              hours = 12;
                              System.out.println(this.toString());
                        }
                        //account for noon
                        else if (hours == 12)
                        {
                              afternoon = true;
                              System.out.println(this.toString());

                        }
                        //morning times don't need converting
                        else
                        {
                              System.out.println(this.toString());
                        }
                  }
            }
      }
```

Code Listing 10.1 continued on next page

```
    /**toString method returns a conventional time
    @return a conventional time with am or pm*/
    public String toString()
    {
         String am_pm;
         String zero = "";
         if (afternoon)
              am_pm = "PM";
         else
              am_pm = "AM";
         if (minutes < 10)
              zero = "0";

         return hours + ":" + zero + minutes + " " + am_pm;
    }
}
```

Code Listing 10.2 (TimedDemo.java)

```
public class TimeDemo
{
    public static void main (String [ ] args)
    {
         Scanner keyboard = new Scanner(System.in);
         char answer = 'Y';
         String enteredTime;
         String response;

         while (//CHECK ANSWER AFTER CONVERTING TO CAPITAL)
         {
              System.out.print("Enter a miitary time using" +
                   " the ##:## form   ");
              enteredTime = keyboard.nextLine();
              Time now = new Time (enteredTime);
              System.out.println(
                   "Do you want to enter another (Y/N)?  ");
              response = keyboard.nextLine();
              answer = response.charAt(0);
         }

    }
}
```

Code Listing 10.3 (secret.txt)

```
January is the first month and December is the last.  Violet is a
purple color as are lilac and plum.
```

Chapter 11 Lab
Inheritance

Objectives

- Be able to derive a class from an existing class
- Be able to define a class hierarchy in which methods are overridden and fields are hidden
- Be able to use derived-class objects
- Implement a copy constructor

Introduction

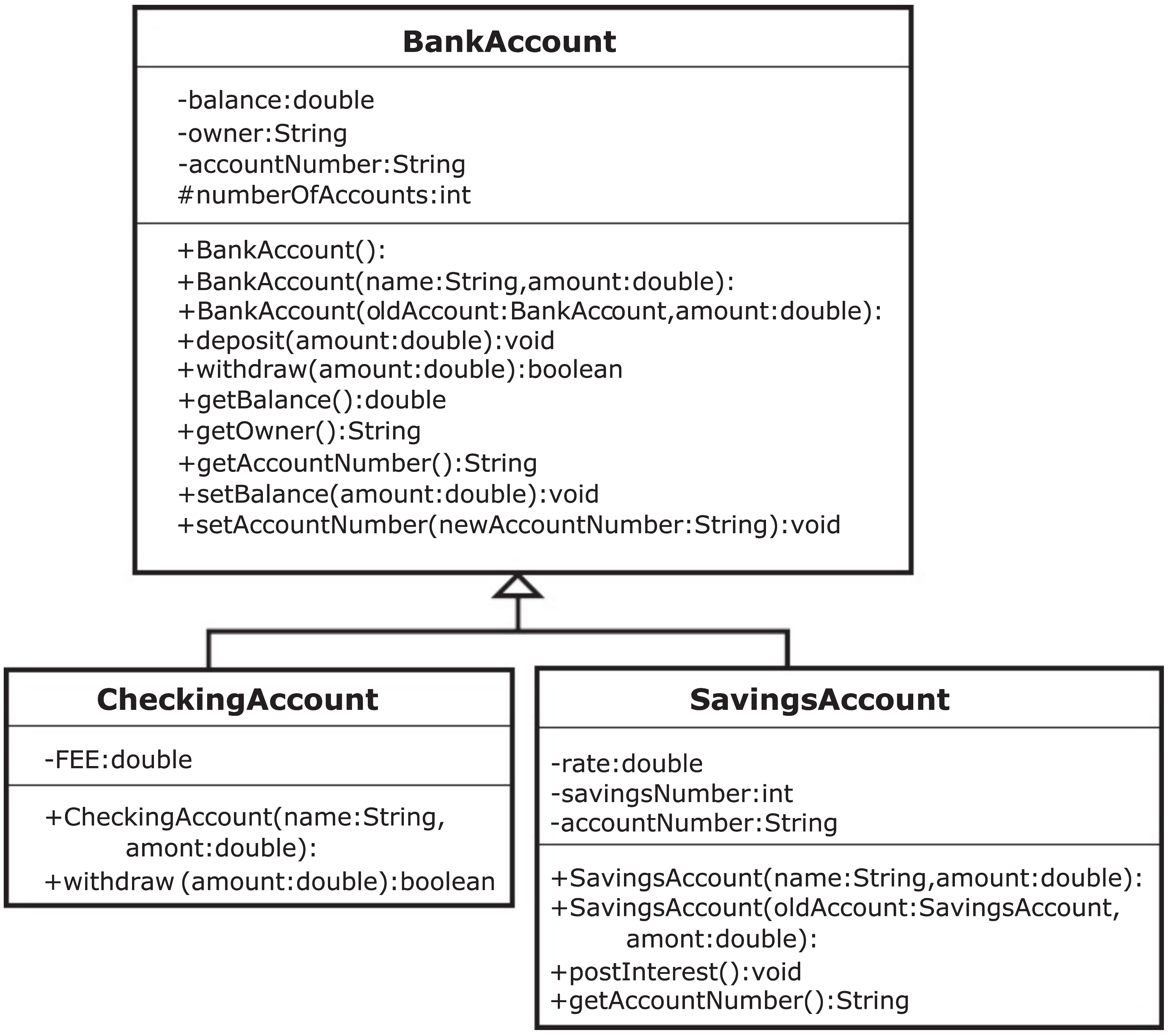
In this lab, you will be creating new classes that are derived from a class called BankAccount. A checking account ***is a*** bank account and a savings account ***is a*** bank account as well. This sets up a relationship called inheritance, where BankAccount is the superclass and CheckingAccount and SavingsAccount are subclasses.

This relationship allows CheckingAccount to inherit attributes from BankAccount (like owner, balance, and accountNumber, but it can have new attributes that are specific to a checking account, like a fee for clearing a check. It also allows CheckingAccount to inherit methods from BankAccount, like deposit, that are universal for all bank accounts.

You will write a withdraw method in CheckingAccount that overrides the withdraw method in BankAccount, in order to do something slightly different than the original withdraw method.

You will use an instance variable called accountNumber in SavingsAccount to hide the accountNumber variable inherited from BankAccount.

The UML diagram for the inheritance relationship is as follows:


BankAccount
-balance:double
-owner:String
-accountNumber:String
#numberOfAccounts:int
+BankAccount():
+BankAccount(name:String,amount:double):
+BankAccount(oldAccount:BankAccount,amount:double):
+deposit(amount:double):void
+withdraw(amount:double):boolean
+getBalance():double
+getOwner():String
+getAccountNumber():String
+setBalance(amount:double):void
+setAccountNumber(newAccountNumber:String):void
CheckingAccount
-FEE:double
+CheckingAccount(name:String, amont:double):
+withdraw (amount:double):boolean
SavingsAccount
-rate:double
-savingsNumber:int
-accountNumber:String
+SavingsAccount(name:String,amount:double):
+SavingsAccount(oldAccount:SavingsAccount, amont:double):
+postInterest():void
+getAccountNumber():String

Task #1 Extending BankAccount

1. Copy the files *AccountDriver.java* (code listing 11.1) and *BankAccount.java* (code listing 11.2) from www.aw.com/cssupport or as directed by your instructor. *BankAccount.java* is complete and will not need to be modified.
2. Create a new class called **CheckingAccount** that **extends BankAccount**.
3. It should contain a static constant **FEE** that represents the cost of clearing one check. Set it equal to 15 cents.
4. Write a constructor that takes a name and an initial amount as parameters. It should call the constructor for the superclass. It should initialize `accountNumber` to be the current value in `accountNumber` concatenated with –10 (All checking accounts at this bank are identified by the extension –10). There can be only one checking account for each account number. Remember since `accountNumber` is a private member in BankAccount, it must be changed through a mutator method.
5. Write a new instance method, **withdraw**, that overrides the withdraw method in the superclass. This method should take the amount to withdraw, add to it the fee for check clearing, and call the withdraw method from the superclass. Remember that to override the method, it must have the same method heading. Notice that the withdraw method from the superclass returns true or false depending if it was able to complete the withdrawal or not. The method that overrides it must also return the same true or false that was returned from the call to the withdraw method from the superclass.
6. Compile and debug this class.

Task #2 Creating a Second Subclass

1. Create a new class called **SavingsAccount** that **extends BankAccount**.
2. It should contain an instance variable called `rate` that represents the annual interest rate. Set it equal to 2.5%.
3. It should also have an instance variable called `savingsNumber`, initialized to 0. In this bank, you have one account number, but can have several savings accounts with that same number. Each individual savings account is identified by the number following a dash. For example, 100001-0 is the first savings account you open, 100001-1 would be another savings account that is still part of your same account. This is so that you can keep some funds separate from the others, like a Christmas club account.
4. An instance variable called `accountNumber` that will hide the `accountNumber` from the superclass, should also be in this class.
5. Write a constructor that takes a name and an initial balance as parameters and calls the constructor for the superclass. It should initialize `accountNumber` to be the current value in the superclass `accountNumber` (the hidden instance variable) concatenated with a hyphen and then the savingsNumber.
6. Write a method called **postInterest** that has no parameters and returns no value. This method will calculate one month's worth of interest on the balance and deposit it into the account.
7. Write a method that overrides the **getAccountNumber** method in the superclass.
8. Write a copy constructor that creates another savings account for the same person. It should take the original savings account and an initial balance as parameters. It should call the copy constructor of the superclass, assign the `savingsNumber` to be one more than the `savingsNumber` of the original savings account. It should assign the `accountNumber` to be the `accountNumber` of the superclass concatenated with the hypen and the `savingsNumber` of the new account.
9. Compile and debug this class.
10. Use the AccountDriver class to test out your classes. If you named and created your classes and methods correctly, it should not have any difficulties. If you have errors, do not edit the AccountDriver class. You must make your classes work with this program.
11. Running the program should give the following output:

```
Account Number 100001-10 belonging to Benjamin Franklin
Initial balance = $1000.00
After deposit of $500.00,  balance = $1500.00
After withdrawal of $1000.00,  balance = $499.85
```

```
Account Number 100002-0 belonging to William Shakespeare
Initial balance = $400.00
After deposit of $500.00,  balance = $900.00
Insuffient funds to withdraw $1000.00,  balance = $900.00
After monthly interest has been posted, balance = $901.88

Account Number 100002-1 belonging to William Shakespeare
Initial balance = $5.00
After deposit of $500.00, balance = $505.00
Insuffient funds to withdraw $1000.00,  balance = $505.00

Account Number 100003-10 belonging to Isaac Newton
```

Code Listing 11.1 (AccountDriver.java)

```
import java.text.*;              // to use Decimal Format
/**Demonstrates the BankAccount and derived classes*/
public class AccountDriver
{
    public static void main(String[] args)
    {
         double put_in = 500;
         double take_out = 1000;

         DecimalFormat myFormat;
         String money;
         String money_in;
         String money_out;
         boolean completed;

         // to get 2 decimals every time
         myFormat = new DecimalFormat("#.00");

         //to test the Checking Account class
         CheckingAccount myCheckingAccount =
              new CheckingAccount ("Ben Franklin", 1000);
         System.out.println ("Account Number "
              + myCheckingAccount.getAccountNumber() +
              " belonging to " + myCheckingAccount.getOwner());
         money  = myFormat.format(
              myCheckingAccount.getBalance());
         System.out.println ("Initial balance = $" + money);
         myCheckingAccount.deposit (put_in);
         money_in = myFormat.format(put_in);
         money  = myFormat.format(
              myCheckingAccount.getBalance());
         System.out.println ("After deposit of $" + money_in
              + ",  balance = $" + money);
         completed = myCheckingAccount.withdraw(take_out);
         money_out = myFormat.format(take_out);
```

Code Listing 11.1 continued on next page.

```
money   = myFormat.format(
      myCheckingAccount.getBalance());
if (completed)
{
      System.out.println ("After withdrawal of $" +
            money_out    + ",  balance = $" + money);
}
else
{
      System.out.println ("Insuffient funds to " +
            " withdraw $" + money_out    +
            ",  balance = $" + money);
}
System.out.println();

//to test the savings account class
SavingsAccount yourAccount =
      new SavingsAccount ("William Shakespeare", 400);
System.out.println ("Account Number "
      + yourAccount.getAccountNumber() +
      " belonging to " + yourAccount.getOwner());
money   = myFormat.format(yourAccount.getBalance());
System.out.println ("Initial balance = $" + money);
yourAccount.deposit (put_in);
money_in = myFormat.format(put_in);
money   = myFormat.format(yourAccount.getBalance());
System.out.println ("After deposit of $" + money_in
      + ",  balance = $" + money);
completed = yourAccount.withdraw(take_out);
money_out = myFormat.format(take_out);
money   = myFormat.format(yourAccount.getBalance());
if (completed)
{
      System.out.println ("After withdrawal of $" +
            money_out    + ",  balance = $" + money);
}
else
{
```

Code Listing 11.1 continued on next page

```
        System.out.println ("Insuffient funds to " +
              "withdraw $" + money_out    +
              ",  balance = $" + money);
        }
        yourAccount.postInterest();
        money  = myFormat.format(yourAccount.getBalance());
        System.out.println ("After monthly interest " +
        "has been posted," + "balance = $"    + money);
        System.out.println();

        // to test the copy constructor of the savings account
        //class
        SavingsAccount secondAccount =
              new SavingsAccount (yourAccount,5);
        System.out.println ("Account Number "
              + secondAccount.getAccountNumber()+
              " belonging to " +
              secondAccount.getOwner());
        money = myFormat.format(secondAccount.getBalance());
        System.out.println ("Initial balance = $" + money);
        secondAccount.deposit (put_in);
        money_in = myFormat.format(put_in);
        money = myFormat.format(secondAccount.getBalance());
        System.out.println ("After deposit of $" + money_in
              + ", balance = $" + money);
        secondAccount.withdraw(take_out);
        money_out = myFormat.format(take_out);
        money = myFormat.format(secondAccount.getBalance());
        if (completed)
        {
              System.out.println ("After withdrawal of $" +
                    money_out + ",  balance = $" + money);
        }
        else
        {
```

Code Listing 11.1 continued on next page

```
                System.out.println ("Insuffient funds to " +
                        "withdraw $" + money_out   +
                        ",  balance = $" + money);
        }
        System.out.println();

        //to test to make sure new accounts are numbered
        //correctly
        CheckingAccount yourCheckingAccount =
                new CheckingAccount ("Isaac Newton", 5000);
        System.out.println ("Account Number "
                + yourCheckingAccount.getAccountNumber()
                + " belonging to "
                + yourCheckingAccount.getOwner());
    }
}
```

Code Listing 11.2 (BankAccount.java)

```
/**Defines any type of bank account*/
public abstract class BankAccount
{
    /**class variable so that each account has a unique
    number*/
    protected static int numberOfAccounts = 100001;

    /**current balance in the account*/
    private double balance;
    /** name on the account*/
    private String owner;
    /** number bank uses to identify account*/
    private String accountNumber;

    /**default constructor*/
    public BankAccount()
    {
         balance = 0;
         accountNumber = numberOfAccounts + "";
         numberOfAccounts++;
    }

    /**standard constructor
    @param name the owner of the account
    @param amount the beginning balance*/
    public BankAccount(String name, double amount)
    {
         owner = name;
         balance = amount;
         accountNumber = numberOfAccounts + "";
         numberOfAccounts++;
    }

    /**copy constructor creates another account for the same
    owner
    @param oldAccount the account with information to copy
```

Code Listing 11.2 continued on next page.

```
@param the beginning balance of the new account*/
public BankAccount(BankAccount oldAccount, double amount)
{
     owner = oldAccount.owner;
     balance = amount;
     accountNumber = oldAccount.accountNumber;
}

/**allows you to add money to the account
@param amount the amount to deposit in the account*/
public void deposit(double amount)
{
     balance = balance + amount;
}

/**allows you to remove money from the account if
enough money is available,returns true if the transaction
was completed, returns false if the there was not enough
money.
@param amount  the amount to withdraw from the account
@return true if there was sufficient funds to complete
the transaction, false otherwise*/
public boolean withdraw(double amount)
{
     boolean completed = true;

     if (amount <= balance)
     {
          balance = balance - amount;
     }
     else
     {
          completed = false;
     }
     return completed;
}
```

Code Listing 11.2 continued on next page

```
    /**accessor method to balance
    @return the balance of the account*/
    public double getBalance()
    {
         return balance;
    }

    /**accessor method to owner
    @return the owner of the account*/
    public String getOwner()
    {
         return owner;
    }

    /**accessor method to account number
    @return the account number*/
    public String getAccountNumber()
    {
         return accountNumber;
    }

    /**mutator method to change the balance
    @param newBalance the new balance for the account*/
    public void setBalance(double newBalance)
    {
         balance = newBalance;
    }

    /**mutator method to change the account number
    @param newAccountNumber the new account number*/
    public void setAccountNumber(String newAccountNumber)
    {
         accountNumber = newAccountNumber;
    }
}
```

Chapter 12 Lab

Exceptions and I/O Streams

Objectives

- Be able to write code that handles an exception
- Be able to write code that throws an exception
- Be able to write a custom exception class

Introduction

This program will ask the user for a person's name and social security number. The program will then check to see if the social security number is valid. An exception will be thrown if an invalid SSN is entered.

You will be creating your own exception class in this program. You will also create a driver program that will use the exception class. Within the driver program, you will include a **static** method that throws the exception. Note: Since you are creating all the classes for this lab, there are no files on www.aw.com/cssupport.

Task #1 Writing a Custom Exception Class

1. Create an exception class called SocSecException. The UML for this class is below.

SocSecException
+SocSecException(String error):

The constructor will call the superclass constructor. It will set the message associated with the exception to "Invalid social security number" concatenated with the error string.

2. Create a driver program called *SocSecProcessor.java*. This program will have a main method and a static method called isValid that will check if the social security number is valid.

SocSecProcessor
+main(args:String[ ]):void +isValid(ssn:String):boolean

Task #2 Writing Code to Handle an Exception

1. In the main method:
   a) The main method should read a name and social security number from the user as Strings.
   b) The main method should contain a try-catch statement. This statement tries to check if the social security number is valid by using the method isValid. If the social security number is valid, it prints the name and social security number. If a SocSecException is thrown, it should catch it and print out the name, social security number entered, and an associated error message indicating why the social security number is invalid.
   c) A loop should be used to allow the user to continue until the user indicates that they do not want to continue.
2. The **static** isValid method:
   a) This method throws a SocSecException.
   b) True is returned if the social security number is valid, false otherwise.
   c) The method checks for the following errors and throws a SocSecException with the appropriate message.
      i) Number of characters not equal to 11. (Just check the length of the string)
      ii) Dashes in the wrong spots.
      iii) Any non-digits in the SSN.
      iv) Hint: Use a loop to step through each character of the string, checking for a digit or hyphen in the appropriate spots.
3. Compile, debug, and run your program. Sample output is shown below with user input in bold.

   OUTPUT (boldface is user input)

```
Name?  Sam Sly
SSN?   333-00-999
Invalid the social security number, wrong number of charac-
ters
Continue?  y
Name?  George Washington
SSN?   123-45-6789
George Washington 123-45-6789 is valid
Continue?  y
Name?  Dudley Doright
SSN?   222-00-999o
Invalid the social security number, contains a character
that is not a digit
Continue?  y
```

```
Name?  Jane Doe
SSN?   333-333-333
Invalid the social security number, dashes at wrong positions
Continue?  n
```

Chapter 13 Lab
Advanced GUI Applications

Objectives

- Be able to add a menu to the menu bar
- Be able to use nested menus
- Be able to add scroll bars, giving the user the option of when they will be seen.
- Be able to change the look and feel, giving the user the option of which look and feel to use

Introduction

In this lab we will be creating a simple note taking interface. It is currently a working program, but we will be adding features to it. The current program displays a window which has one item on the menu bar, **Notes**, which allows the user 6 choices. These choices allow the user to store and retrieve up to 2 different notes. It also allows the user to clear the text area or exit the program.

We would like to add features to this program which allows the user to change how the user interface appears. We will be adding another choice on the menu bar called **Views**, giving the user choices about scroll bars and the look and feel of the GUI.

Task #1 Creating a Menu with Submenus

1. Copy the file *NoteTaker.java* (code listing 13.1) from www.aw.com/cssuport or as directed by your instructor.
2. Compile and run the program. Observe the horizontal menu bar at the top which has only one menu choice, **Notes**. We will be adding an item to this menu bar called **Views** that has two submenus. One named **Look and Feel** and one named **Scroll Bars**. The submenu named **Look and Feel** lets the user change the look and feels: **Metal**, **Motif**, and **Windows**. The submenu named **Scroll Bars** offers the user three choices: **Never**, **Always**, and **As Needed**. When the user makes a choice, the scroll bars are displayed according to the choice.
3. We want to logically break down the problem and make our program easier to read and understand. We will write separate methods to create each of the vertical menus. The three methods that we will be writing are **`createViews()`**, **`createScrollBars()`**, and **`createLookAndFeel()`**. The method headings with empty bodies are provided.
4. Let's start with the **`createLookAndFeel()`** method. This will create the first submenu shown in figure 1. There are three items on this menu, **Metal**, **Motif**, and **Windows**. We will create this menu by doing the following:
   a) Create a new JMenu with the name **Look and Feel**.
   b) Create a new JMenuItem with the name **Metal**.
   c) Add an action listener to the menu item (see the **`createNotes()`** method to see how this is done).
   d) Add the menu item to the menu.
   e) Repeat steps b through d for each of the other two menu items.
5. Similarly, write the **`createScrollBars()`** method to create a JMenu that has three menu items, **Never**, **Always**, and **As Needed**. See figure 2.
6. Now that we have our submenus, these menus will become menu items for the **Views** menu. The **`createViews()`** method will make the vertical menu shown cascading from the menu choice **Views** as shown in figure. We will do this as follows
   a) Create a new JMenu with the name **Views**.
   b) Call the **`createLookAndFeel()`** method to create the **Look and Feel** submenu.
   c) Add an action listener to the **Look and Feel** menu.
   d) Add the look and feel menu to the **Views** menu.
   e) Repeat steps b through d for the **Scroll Bars** menu item, this time calling the **`createScrollBars()`** method.
7. Finish creating your menu system by adding the **Views** menu to the menu bar in the constructor.

Task #2 Adding Scroll Bars and Editing the Action Listener

1. Add scroll bars to the text area by completing the following steps in the constructor
   a) Create a JScrollPane object called **`scrolledText`**, passing in **`theText`**.
   b) Change the line that adds to the textPanel, by passing in **`scrolledText`** (which now has **`theText`**.)
2. Edit the action listener by adding 6 more branches to the else-if logic. Each branch will compare the actionCommand to the 6 submenu items: **Metal**, **Motif**, **Window**, **Never**, **Always**, and **As Needed**.
   a) Each **Look and Feel** submenu item will use a try-catch statement to set the look and feel to the appropriate one, displaying an error message if this was not accomplished.
   b) Each **Scroll Bars** submenu item will set the horizontal and vertical scroll bar policy to the appropriate values.
   c) Any components that have already been created need to be updated. This can be accomplished by calling the **SwingUtilities.updateComponentTreeUI** method, passing a reference to the component that you want to update as an argument. Specifically you will need to add the line

```
SwingUtilities.updateComponentTreeUIgetContentPane());
```

   to each branch that you just added to the logic structure.

Figure 1

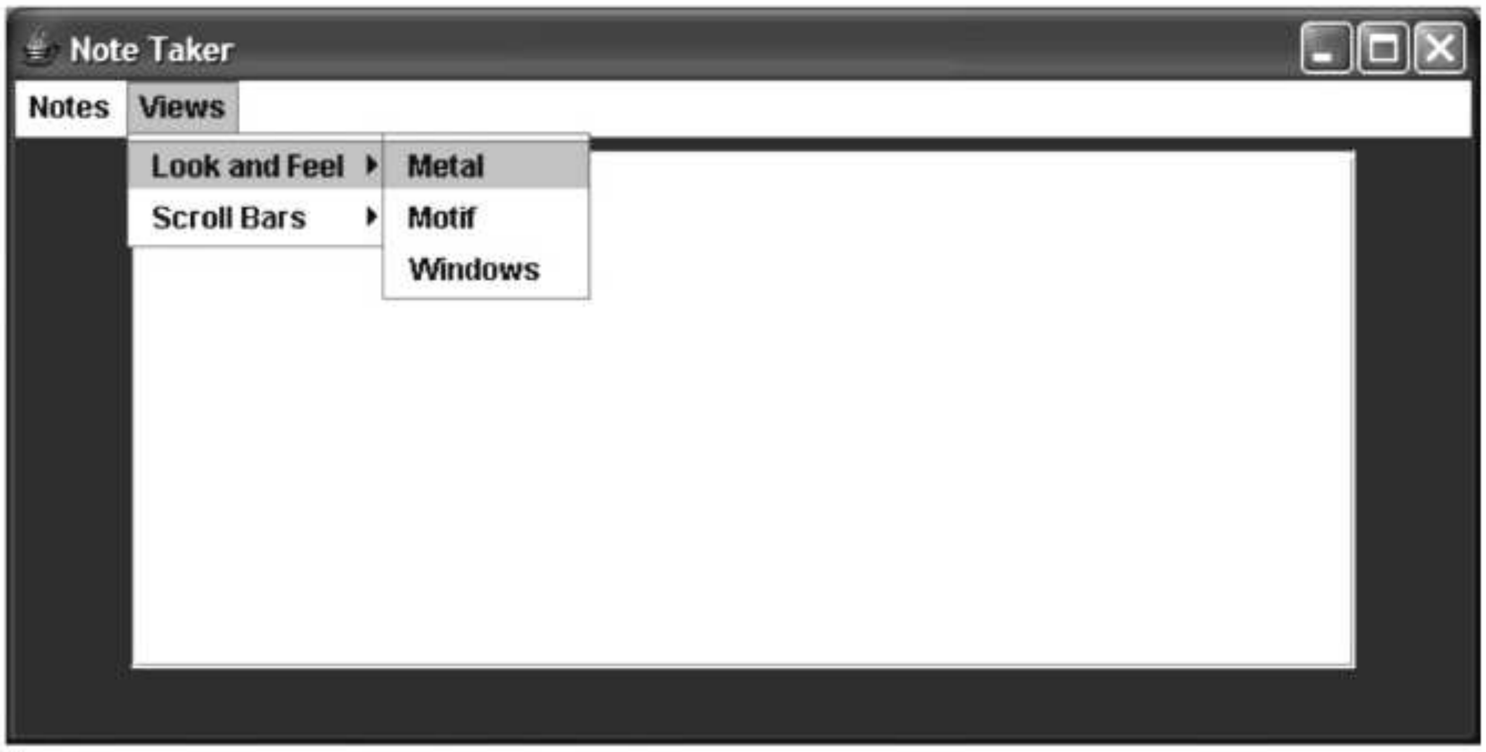


Figure 2

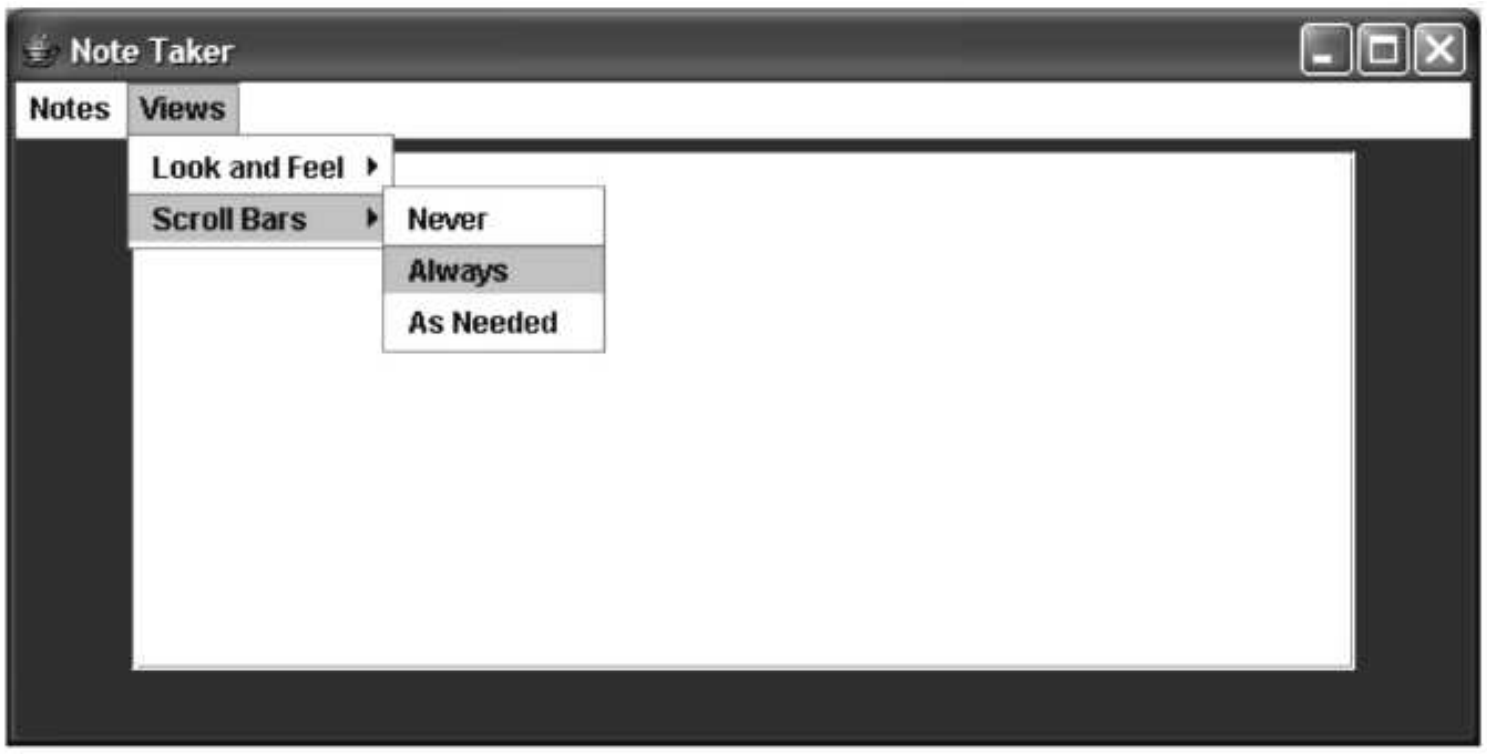
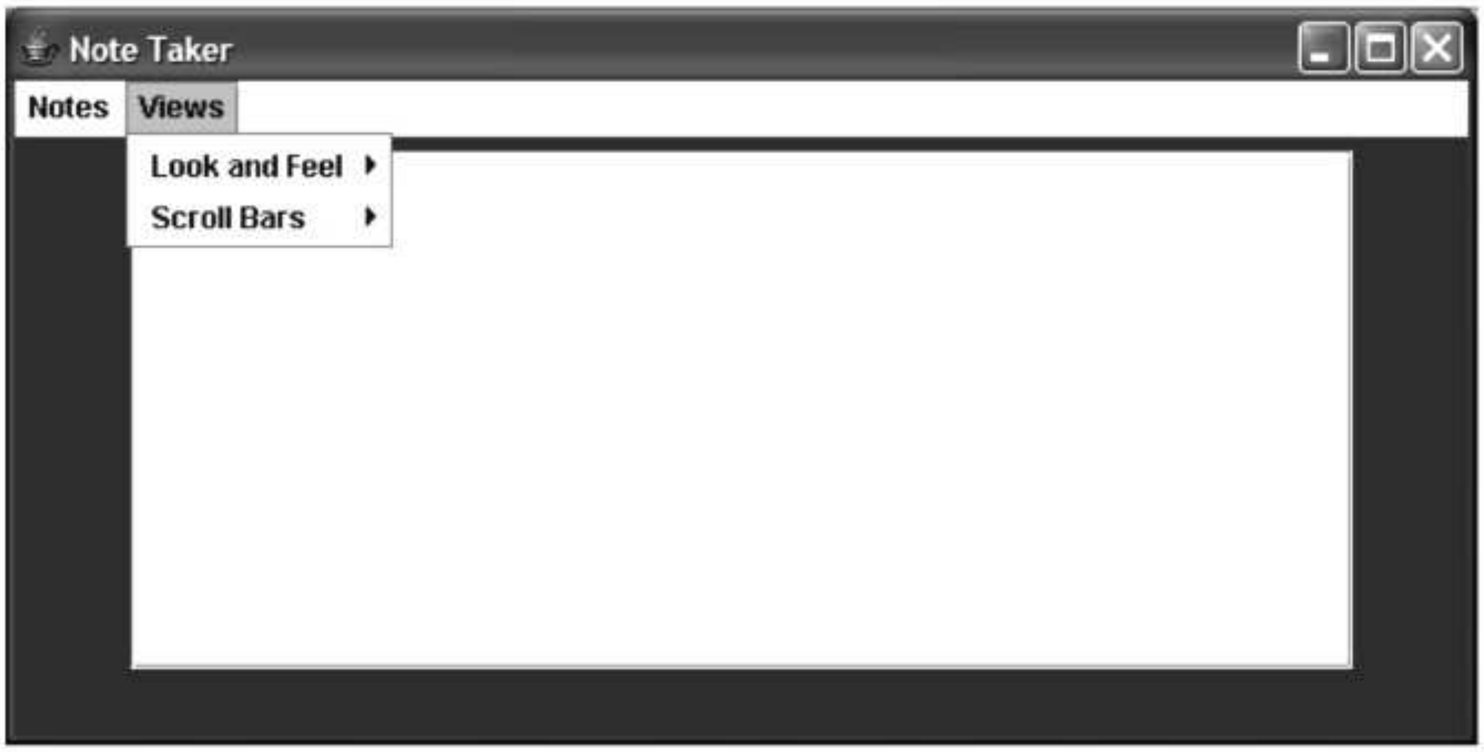


Figure 3



Code Listing 13.1 (NoteTaker.java)

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class NoteTaker extends JFrame
{
   //constants for set up of note taking area
   public static final int WIDTH = 600;
   public static final int HEIGHT = 300;
   public static final int LINES = 13;
   public static final int CHAR_PER_LINE = 45;

   //objects in GUI
   private JTextArea theText;  //area to take notes
   private JMenuBar mBar;   //horizontal menu bar
   private JPanel textPanel;   //panel to hold scrolling text area
   private JMenu notesMenu;    //vertical menu with choices for notes

   //****THESE ITEMS ARE NOT YET USED.
   //****YOU WILL BE CREATING THEM IN THIS LAB
   private JMenu viewMenu;   //vertical menu with choices for views
   private JMenu lafMenu;   //vertical menu with look and feel
   private JMenu sbMenu;   //vertical menu with scroll bar option
   private JScrollPane scrolledText;//scroll bars

   //default notes
   private String note1 = "No Note 1.";
   private String note2 = "No Note 2.";

   /**constructor*/
   public NoteTaker()
   {
      //create a closeable JFrame with a specific size
      super("Note Taker");
      setSize(WIDTH, HEIGHT);
      setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
```
Code Listing 13.1 continued on next page.

```
      //get contentPane and set layout of the window
      Container contentPane = getContentPane();
      contentPane.setLayout(new BorderLayout());

      //creates the vertical menus
      createNotes();
      createViews();

      //creates horizontal menu bar and
      //adds vertical menus to it
      mBar = new JMenuBar();
      mBar.add(notesMenu);
      //****ADD THE viewMenu TO THE MENU BAR HERE
      setJMenuBar(mBar);

      //creates a panel to take notes on
      textPanel = new JPanel();
      textPanel.setBackground(Color.blue);
      theText = new JTextArea(LINES, CHAR_PER_LINE);
      theText.setBackground(Color.white);
      //****CREATE A JScrollPane OBJECT HERE CALLED scrolledText
      //****AND PASS IN theText, THEN
      //****CHANGE THE LINE BELOW BY PASSING IN scrolledText
      textPanel.add(theText);
      contentPane.add(textPanel, BorderLayout.CENTER);
   }

   /**creates vertical menu associated with Notes
   menu item on menu bar*/
   public void createNotes()
   {
      notesMenu = new JMenu("Notes");
      JMenuItem item;

      item = new JMenuItem("Save Note 1");
      item.addActionListener(new MenuListener());
      notesMenu.add(item);
```

Code Listing 13.1 continued on next page.

```
      item = new JMenuItem("Save Note 2");
      item.addActionListener(new MenuListener());
      notesMenu.add(item);

      item = new JMenuItem("Open Note 1");
      item.addActionListener(new MenuListener());
      notesMenu.add(item);

      item = new JMenuItem("Open Note 2");
      item.addActionListener(new MenuListener());
      notesMenu.add(item);

      item = new JMenuItem("Clear");
      item.addActionListener(new MenuListener());
      notesMenu.add(item);

      item = new JMenuItem("Exit");
      item.addActionListener(new MenuListener());
         notesMenu.add(item);
   }

   /**creates vertical menu associated with Views
   menu item on the menu bar*/
   public void createViews()
   {

   }

   /**creates the look and feel submenu*/
   public void createLookAndFeel()
   {

   }

   /**creates the scroll bars submenu*/
   public void createScrollBars()
   {
```

Code Listing 13.1 continued on next page.

```
   }

   private class MenuListener implements ActionListener
   {

        public void actionPerformed(ActionEvent e)
      {
         String actionCommand = e.getActionCommand();
         if (actionCommand.equals("Save Note 1"))
            note1 = theText.getText();
         else if (actionCommand.equals("Save Note 2"))
            note2 = theText.getText();
         else if (actionCommand.equals("Clear"))
            theText.setText("");
         else if (actionCommand.equals("Open Note 1"))
            theText.setText(note1);
         else if (actionCommand.equals("Open Note 2"))
            theText.setText(note2);
         else if (actionCommand.equals("Exit"))
            System.exit(0);
         //****ADD 6 BRANCHES TO THE ELSE-IF STRUCTURE
            //****TO ALLOW ACTION TO BE PERFORMED FOR EACH
            //****MENU ITEM YOU HAVE CREATED
         else
            theText.setText("Error in memo interface");
      }
   }

   public static void main(String[] args)
   {
      NoteTaker gui = new NoteTaker();
      gui.setVisible(true);
   }
}
```

Chapter 14 Lab
Applets and More

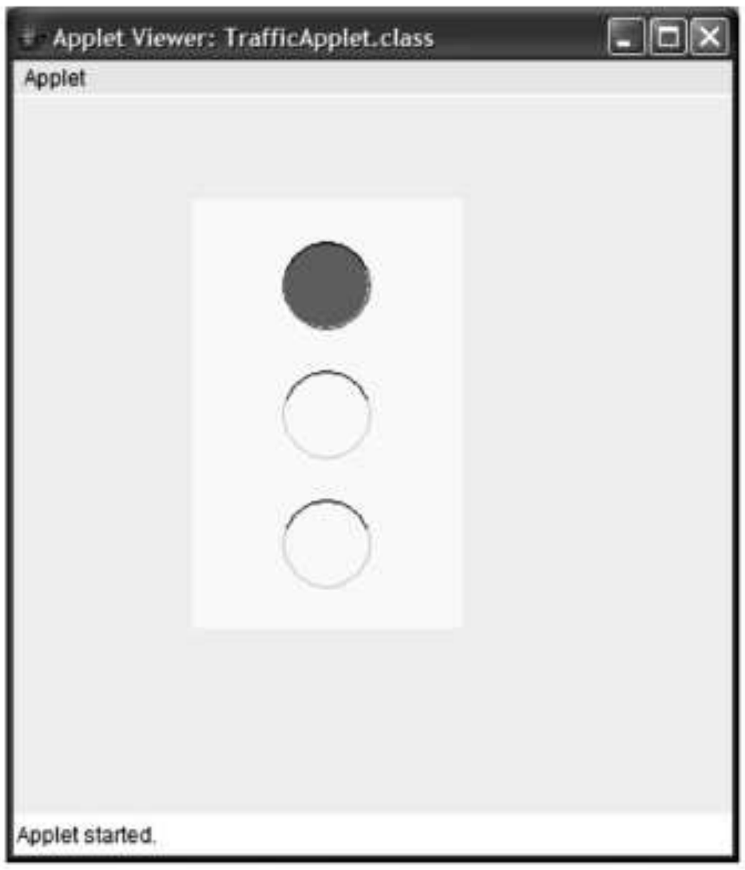
Objectives

- Be able to write an applet
- Be able to draw rectangles, circles, and arcs
- Be able to override the paint method
- Be able to use a timer

Introduction

In this lab we will create an applet that changes the light on a traffic signal. The applet that you create will draw the outside rectangle of a traffic signal and fill it in with yellow. Then it will draw three circles in one column, to resemble the red, orange, and green lights on the traffic signal. Only one circle at a time will be filled in. It will start will green and cycle through the orange, red, and back to green to start the cycle again. However, unlike a traffic signal, each light will remain on for the same amount of time. To accomplish this cycle, we will use a timer object.

When you have finished your applet should appear as shown in figure 1, but with the filled in circle cycling up from green to orange to red and starting over in a continuous changing of the traffic light.



Task #1 Create an Applet

1. Copy the file *TrafficApplet.java* (see code listing 14.1) from www.aw.com/cssupport or as directed by your instructor.
2. This class currently has all the constants you will need to be able to you're your traffic signal. It doesn't have anything else. You will need to change the class heading so that it extends JApplet.

Task #2 The Timer

1. An applet does not have a constructor or a main method. Instead, it has a method named **`init`** that performs the same operations as a constructor. The **`init`** method accepts no arguments and has a void return type. Write an **`init`** method.
2. Inside the **`init`** method, create a timer object passing in the TIME_DELAY constant and a new TimerListener (We will be creating the listener class next).
3. Call the start method with the timer object to generate action events.

Task #3 The TimerListener Class

1. Write a private inner class called TimerListener which implements ActionListener.
2. Inside this class, write an actionPerformed method. This method will check the status variable to see whether it is currently red, orange, or green. Since we want the lights to cycle as a traffic signal, we need to cycle in the order: green, orange, red, green, orange, red, … Once the status is determined, the status should then be set to the next color in the cycle.
3. Redisplay the graphics components (to be created next) by calling the **`repaint`** method.

Task #4 Drawing Graphics

1. Draw the traffic signal by overriding the paint method. For all graphics, use the named constants included in the class.
2. Call the method that is being overridden in the parent class.
3. Create a yellow rectangle (solid color) for the traffic signal. The constants X_TRAFFICLIGHT, Y_TRAFFICLIGHT, TRAFFICLIGHT_WIDTH, and TRAFFICLIGHT_HEIGHT have already been defined for your use.
4. Create round lights of red, orange, and green for the signals. These should be outlines of these colors. The constants X_LIGHTS, Y_REDLIGHT, Y_GREENLIGHT, Y_ORANGELIGHT, and LIGHT_DIAMETER, have already been defined for your use. Only one light will be filled in at a time, when the status indicates that one has been chosen. You will need to check the status to determine which light to fill in. Remember, the status is changed only in the actionPerformed method (already defined) where the **`repaint`** method is also called.
5. Put the shade hoods above the lights by drawing black arcs above each light. The constants HOOD_START_ANGLE and HOOD_ANGLE_SWEPT have already been defined for your use.
6. Try out your applet. If time permits, create a web page on which you can display your applet.

Code Listing 14.1 (TrafficApplet.java)

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TrafficApplet
{
    public final int WIDTH = 300;
    public final int HEIGHT = 400;

    public final int X_TRAFFICLIGHT = WIDTH/3;
    public final int Y_TRAFFICLIGHT = HEIGHT/7;
    public final int TRAFFICLIGHT_WIDTH = WIDTH/2;
    public final int TRAFFICLIGHT_HEIGHT = HEIGHT*3/5;
    public final int LIGHT_DIAMETER = TRAFFICLIGHT_HEIGHT/5;
    public final int HOOD_START_ANGLE = 20;
    public final int HOOD_ANGLE_SWEPT = 140;
    public final int X_LIGHTS =
         TRAFFICLIGHT_WIDTH/3 + X_TRAFFICLIGHT;
    public final int Y_REDLIGHT =
         TRAFFICLIGHT_HEIGHT/10 + Y_TRAFFICLIGHT;
    public final int Y_ORANGELIGHT =
         TRAFFICLIGHT_HEIGHT*4/10 + Y_TRAFFICLIGHT;
    public final int Y_GREENLIGHT =
         TRAFFICLIGHT_HEIGHT*7/10 + Y_TRAFFICLIGHT;
    public final int TIME_DELAY = 1000;

    private String status = "green";  //start with the green
                                      //light
    private Timer timer;   //will allow lights to cycle


}
```

Chapter 15 Lab

Recursion

Objectives

- Be able to trace recursive function calls
- Be able to write non-recursive and recursive methods to find geometric and harmonic progressions

Introduction

In this lab we will follow how the computer executes recursive methods, and will write our own recursive method, as well as the iterative equivalent. There are two common progressions in mathematics, the geometric progression and the harmonic progression. The geometric progression is defined as the product of the first ***n*** integers. The harmonic progression is defined as the product of the inverses of the first ***n*** integers. Mathematically, the definitions are as follows

$$\text{Geometric }(n) = \prod_{i=1}^{n} i = i \times \prod_{i=1}^{n-1} i$$

$$\text{Harmonic }(n) = \prod_{i=1}^{n} \frac{1}{i} = \frac{1}{i} \times \prod_{i=1}^{n-1} \frac{1}{i}$$

Let's look at examples. If we use $\boldsymbol{n} = 4$, the geometric progression would be $1 \times 2 \times 3 \times 4 = 24$, and the harmonic progression would be

$$1 \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4} = \frac{1}{24} = 0.04166\ldots$$

Task #1 Tracing recursive methods

1. Copy the file *Recursion.java* (see code listing 15.1) from www.aw.com/cssupport or as directed by your instructor.
2. Run the program to confirm that the generated answer is correct. Modify the factorial method in the following ways:
   a) add these lines above the first 'if' statement
   ```
   int temp;
   System.out.println(
   "Method call – calculating Factorial of: " + n);
   ```
   b) remove this line in the recursive section at the end of the method
   ```
   return (factorial(n-1) *n);
   ```
   c) add these lines in the recursive section
   ```
   temp = factorial(n-1);
   System.out.println(
   "Factorial of: " + (n-1) + " is " + temp);
   return (temp * n);
   ```
3. Rerun the program and note how the recursive calls are built up on the run-time stack and then the values are calculated in reverse order as the run-time stack "unwinds".

Task #2 Writing Recursive and Iterative Versions of a Method

1. Copy the file *Progression.java* (see code listing 15.2) from www.aw.com/cssupport or as directed by your instructor.
2. You need to write **class** (static) methods for an iterative and a recursive version each of the progressions. You will have 4 methods, **`geometricRecursive`**, **`geometricIterative`**, **`harmonicRecursive`**, and **`harmonicIterative`**. Be sure to match them to the method calls in the main method.

Code Listing 15.1 (Recursive.java)

```
public class Recursion
{
    public static void main(String[] args)
    {
        int n = 7;

        //Test out the factorial
        System.out.println(n + " factorial equals ");
        System.out.println(Recursion.factorial(n));
        System.out.println();

    }

    public static int factorial(int n)
    {
        int temp;
        if (n==0)
        {
            return 1;
        }
        else
        {
            return (factorial(n-1) *n);
        }
    }
}
```

Code Listing 15.2 (Progression.java)

```
import java.util.Scanner;

public class Progression
{
    public static void main(String [] args)
    {
         Scanner keyboard = new Scanner (System.in);
         System.out.println("This program will calculate " +
              "the geometric and ");
         System.out.println("harmonic progression for the " +
              "number you enter.");
         System.out.print("Enter an integer that is " +
              "greater than or equal to 1:  ");
         int input = keyboard.nextInt();
         int geomAnswer = geometricRecursive (input);
         double harmAnswer = harmonicRecursive (input);
         System.out.println("Using recursion:");
         System.out.println("The geometric progression of " +
              input + " is " + geomAnswer);
         System.out.println("The harmonic progression of " +
              input + " is " + harmAnswer);

         geomAnswer = geometricIterative (input);
         harmAnswer = harmonicIterative (input);
         System.out.println("Using iteration:");
         System.out.println("The geometric progression of " +
              input + " is " + geomAnswer);
         System.out.println("The harmonic progression of " +
              input + " is " + harmAnswer);
    }
}
```