**Assignment 2 Turn in Sheet Name:**

Lab Questions (**Total 50 Pts. + 53 EXTRA Pts.**)

Big Java, Late Objects / Java for Everyone, 2e

**Chapter Number: 5 Methods**

1) (5 pts.) Write a static method called AreaOfRectangle that is passed two float-point values for the length and width of a rectangle. The method returns the product of the length and width as a double. Comment the method using javadoc conventions. Write a main method that creates the following variables to describe the sides of a rectangle:

double length = 3.4;

double width = 8.4;

The main method should print the length, width, and area of the rectangle.

|  |
| --- |
| public static void main(String[] args)  {  double length = 3.4;  double width = 8.4;  System.out.println("Rectangle: \nWidth: " + width + "\nLength: " + length);  System.out.println("Area of Rectangle is: " + AreaOfRectangle(width, length));  }/\*\*  \* Method which accepts two double variables in order to calculate the area of  \* a rectangle  \* @param input\_Width  \* @param input\_Length  \* @return sum of Length and Width  \*/  public static double AreaOfRectangle(double input\_Width, double input\_Length)  {  double rectangle\_area = input\_Length \* input\_Width;  return rectangle\_area;  }  }  **OUTPUT:**  **Rectangle:**  **Width: 8.4**  **Length: 3.4**  **Area of Rectangle is: 28.56** |

2) (5 pts.) Why does the following code contain a compile-time error?

public class Area

{

public static void main(String[] args)

{

int x = 2;

int y = 3;

System.out.println("x: " + x + " y: " + y + " Sum: " + sum(x,y));

}

/\*\*

Computes the sum of two arguments.

@param a an int operand to be added

@param b another int operand

@return the sum of a and b

\*/

public static int sum(int a, int b)

{

return a + b;

System.out.println("Finished adding...");

}

}

The code will throw and error of ‘unreachable statement’ as System.out.println("Finished adding..."); will never be executed as it happens after a return statement;

3) (5 pts.) Run the following code. The sum method in this program violates a design principle of the book (that methods should not try to modify an argument) when it assigns 5 to a, and 6 to b. Certainly these values are changed; this can be seen by examining the value the sum method returns. But what about arguments x and y? Are their values changed, too? In other words: Do the assignments made in the method body have side effects in the main program?

public class Area

{

public static void main(String[] args)

{

int x = 2;

int y = 3;

System.out.println("x: " + x + " y: " + y + " Sum: " + sum(x, y));

}

/\*\*

Computes the sum of two arguments.

@param a an int operand to be added

@param b another int operand

@return the sum of a and b

\*/

public static int sum(int a, int b)

{

a = 5;

b = 6;

return a + b;

}

}

**No, there is no effect on the variables X and Y, java is always pass-by-value so whatever happens in the method *sum* will stay there. Assigning a to 5 and b to 6 will change those values only for the duration of the method call.**

4.1) (3 pts.) Write a method called makeRow that is passed two arguments: an int n and a String s, and which returns a String containing n copies of s, concatenated in a row. For instance, if we call the method with makeRow(5, "\*"), the method returns \*\*\*\*\*. Write a main method that uses the method to print the string \*\*\*\*\*=====\*\*\*\*\*.

|  |
| --- |
| public static void main(String[] args)  {  makeRow(5,"\*");  makeRow(5,"=");  makeRow(5,"\*");  }    public static void makeRow(int n, String s)  {  for(int i = 0; i < n; i++)  System.out.print(s);  } |

4.2) (3 pts.) Reuse makeRow to write a method called printUpTriangle that is passed two arguments, an int n and a String s. It should print a right triangle in which the base of the triangle is made of n copies of s, and the vertex of the triangle has a single copy of s on the right. For example, calling printUpTriangle(13, "\*"); prints the following lines:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

Write a main method that calls printUpTriangle(13, "\*").

|  |
| --- |
| public static void main(String[] args)  {  printUpTriangle(13, "\*");  }    public static void makeRow(int n, String s)  {  for(int i = 0; i < n; i++)  System.out.print(s);  }    public static void printUpTriangle(int n, String s)  {  for(int i = 1; i <= n; i++)  {  makeRow((n-i)," ");  makeRow(i,s);  System.out.println();    }  }  **OUTPUT:**  **\***  **\*\***  **\*\*\***  **\*\*\*\***  **\*\*\*\*\***  **\*\*\*\*\*\***  **\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\*\*** |

4.3) (3 pts.) Reuse makeRow by writing a method called printDownTriangle that is passed an int n and a String s, and that prints a right triangle in which the base (at the top) of the triangle is made of n copies of s, and the vertex of the triangle has a single copy of s on the right. For example, calling printDownTriangle(13, "\*"); prints the following lines:

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

Write a main method that calls printDownTriangle(13, "\*").

|  |
| --- |
| public static void main(String[] args)  {  printDownTriangle(13, "\*");  }    public static void makeRow(int n, String s)  {  for(int i = 0; i < n; i++)  System.out.print(s);  }    public static void printDownTriangle(int n, String s)  {  for(int i = 0; i <= n; i++)  {  makeRow(i," ");  makeRow((n-i),s);  System.out.println();    }  }  **OUTPUT:**  **\*\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\***  **\*\*\*\*\*\***  **\*\*\*\*\***  **\*\*\*\***  **\*\*\***  **\*\***  **\*** |

4.4) (3 pts.) Write a method called printPyramid that is passed an odd integer n and a String s, and that prints a pyramidal shape using s. The top of the pyramid has a single copy of s, and each successive row has two additional copies of s. The last row contains n copies of s. For example, calling printPyramid(21, "\*"); prints the following lines:

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Test your work by calling printPyramid(21, "\*") from the main method.

|  |
| --- |
| public static void main(String[] args)  {  printPyramid(21, "\*");  }    public static void makeRow(int n, String s)  {  for(int i = 0; i < n; i++)  System.out.print(s);  }    public static void printPyramid(int n, String s)  {    for(int i = 1; i <= (n/2)+1; i++)  {  makeRow((n/2-i)+1," ");  makeRow(i,s);  makeRow(i-1,s);  System.out.println();  }  }  **OUTPUT:**  **\***  **\*\*\***  **\*\*\*\*\***  **\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*** |

4.5) (3 pts.) Build a method called printHouse that is passed an odd integer n and a String s, and that prints a house-like shape using s. The top of the house is a pyramid and the bottom of the house is a square. Reuse the methods you’ve already written. Calling printHouse(13, "\*"); prints the following lines:

\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*

Test your work in a program by calling printHouse(11, "\*").

|  |
| --- |
| public static void main(String[] args)  {  printHouse(11, "\*");  }    public static void makeRow(int n, String s)  {  for(int i = 0; i < n; i++)  System.out.print(s);  }    public static void printPyramid(int n, String s)  {    for(int i = 1; i <= (n/2)+1; i++)  {  makeRow((n/2-i)+1," ");  makeRow(i,s);  makeRow(i-1,s);  System.out.println();  }  }    public static void printHouse(int n, String s)  {  printPyramid(n, s);  for(int i = 1; i < n; i++)  {  makeRow(n, s);  System.out.println();  }  }  **OUTPUT:**  **\***  **\*\*\***  **\*\*\*\*\***  **\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\***  **\*\*\*\*\*\*\*\*\*\*\*** |

5.1) (5 pts.) Credit card numbers contain a check digit that is used to help detect errors and verify that the card number is valid. (You can read about the Luhn algorithm at http://en.wikipedia.org/wiki/Luhn\_algorithm.) The check digit can help detect all single-digit errors and almost all transpositions of adjacent digits. In this problem we write some methods that will allow us to quickly check whether a card number is invalid. We will limit our numbers to seven digits and the rightmost digit will be the check digit. For example, if the credit card number is 2315778, the check digit is 8. We number the digit positions starting at the check digit, moving left. Here’s the numbering for credit card number 2315778:

Position Digit

1 8

2 7

3 7

4 5

5 1

6 3

7 2

In order to verify that the card number is correct we will need to “decode” every digit. The decoding process depends on the position of the digit within the credit card number:

a) If the digit is in an odd-numbered position, simply return the digit,

b) If the digit is in an even-numbered position, double it. If the result is a single digit, return it; otherwise, add the two digits in the number and return the sum.

For example, if we decode 8 and it is in an odd position, we return 8. On the other hand, if 8 is in an even position, we double it to get 16, and then return 1 + 6 = 7. Decoding 4 in an odd position would return 4, and decoding it an even position would return 8.

As a first step to being able to being able to detect invalid numbers, you should write a method called decode that is passed an int for the digit and a boolean for the position (true = even position, false = odd position). The method should decode the digit using the method described above and return an int. Test your method with the main method below:

public class Luhn

{

public static void main(String[] args)

{

boolean even = false;

System.out.println(decode(1, even));

System.out.println(decode(2, even));

System.out.println(decode(3, even));

System.out.println(decode(4, even));

System.out.println(decode(5, even));

System.out.println(decode(6, even));

System.out.println(decode(7, even));

System.out.println(decode(8, even));

System.out.println(decode(9, even));

even = ! even;

System.out.println(decode(1, even));

System.out.println(decode(2, even));

System.out.println(decode(3, even));

System.out.println(decode(4, even));

System.out.println(decode(5, even));

System.out.println(decode(6, even));

System.out.println(decode(7, even));

System.out.println(decode(8, even));

System.out.println(decode(9, even));

}

public static int decode(int digit, boolean even)

{

if(!even){

return digit;}

else

{

int double\_digit = 2 \* digit;

if(double\_digit < 10)

return double\_digit;

else

return (double\_digit%10)+1;

}

}

5.2. (5 pts.) Now that we can decode single digits, it’s time to build some code that will help detect errors in credit card numbers. Here’s the idea:

a) Starting with the check digit and moving left, compute the sum of all the decoded digits.

b) Compute the remainder of the sum using integer division by 10. If the result is not zero, the credit card number is invalid. Otherwise, the card number is likely to be valid.

Here are two examples:

Card number: 2315778 Card number 1234567

decode(8, false) = 8 decode(7, false) = 7

decode(7, true) = 5 decode(6, true) = 3

decode(7, false) = 7 decode(5, false) = 5

decode(5, true) = 1 decode(4, true) = 8

decode(1, false) = 1 decode(3, false) = 3

decode(3, true) = 6 decode(2, true) = 4

decode(2, false) = 2 decode(1, false) = 1

Sum = 30 Sum = 31

30 mod 10 = 0 31 mod 10 = 1

This number may be valid This number is invalid

Write a static method called checkDigits that is passed a seven-digit credit card number and that performs the steps described above. Reuse the decode method that you wrote in Lab 5.1. The method should return the word “valid” if the number passes the test and “invalid” otherwise.

Test your methods with the main method below:

public class Luhn

{

public static void main(String[] args)

{

int num = 2315778;

System.out.println("Credit card number: " + num + " is " + checkDigits(num));

num = 1234567;

System.out.println("Credit card number: " + num + " is " + checkDigits(num));

num = 7654321;

System.out.println("Credit card number: " + num + " is " + checkDigits(num));

num = 1111111;

System.out.println("Credit card number: " + num + " is " + checkDigits(num));

}

public static String checkDigits(int cNum)

{

int cardSum = 0, currentDigit;

for(int i = 1; i < 8; i++)

{

currentDigit = cNum % 10;

cNum = cNum / 10;

if(i % 2 == 0)

cardSum += decode(currentDigit, true);

else

cardSum += decode(currentDigit, false);

}

if ((cardSum % 10) == 0)

return "Valid";

else

return "Invalid";

}

}

**OUTPUT:**

**Credit card number: 2315778 is Valid**

**Credit card number: 1234567 is Invalid**

**Credit card number: 7654321 is Invalid**

**Credit card number: 1111111 is Valid**

6) (5 pts.) A computer’s memory is made up of a sequential collection of bytes where each byte consists of eight bits. When examining the contents of memory, it is sometimes useful to display the contents in hexadecimal (base 16). Conveniently, a single hexadecimal digit can represent four of the bits in a byte:

Binary Hex

0000 0

0001 1

0010 2

0011 3

0100 4

0101 5

0110 6

0111 7

1000 8

1001 9

1010 A

1011 B

1100 C

1101 D

1110 E

1111 F

Write a function called bitsToHex that is passed a byte with an integer value from 0 to 15 and returns a String that contains a hex digit equivalent to the passed value. For example, if the byte contains a decimal 12, the method returns the String "D", and if the byte contains a decimal 9, the method returns the String "9". A byte is the smallest unit of storage in Java. By limiting ourselves to the range 0 to 15, we know that the leftmost four bits in the byte are all 0’s. So, if we create a byte with value 13, the contents of the byte in binary is 00001101 and our bitsToHex method will return "D".

|  |
| --- |
| public static void main(String [] args)  {  for(int i = 0; i < 16; i++)  {  byte my\_byte = (byte)i;  System.out.println( "Byte: " + my\_byte + " is " +  bitsToHex(my\_byte) + " in hexadecimal");  }  }    public static String bitsToHex(byte input\_byte)  {  String toHex[] ={"0", "1", "2","3", "4", "5", "6", "7", "8", "9",  "A", "B", "C", "D", "E", "F"};  return toHex[input\_byte];  }  **OUTPUT:**  **Byte: 0 is 0 in hexadecimal**  **Byte: 1 is 1 in hexadecimal**  **Byte: 2 is 2 in hexadecimal**  **Byte: 3 is 3 in hexadecimal**  **Byte: 4 is 4 in hexadecimal**  **Byte: 5 is 5 in hexadecimal**  **Byte: 6 is 6 in hexadecimal**  **Byte: 7 is 7 in hexadecimal**  **Byte: 8 is 8 in hexadecimal**  **Byte: 9 is 9 in hexadecimal**  **Byte: 10 is A in hexadecimal**  **Byte: 11 is B in hexadecimal**  **Byte: 12 is C in hexadecimal**  **Byte: 13 is D in hexadecimal**  **Byte: 14 is E in hexadecimal**  **Byte: 15 is F in hexadecimal** |

**Chapter Number: 8 Objects and Classes**

7.1) (2 pts.) Object-oriented languages like Java are designed to make it easy for programmers to implement software versions of real-world objects. In learning Java, an important skill to master is the ability to represent an object in code. Objects that we model are described using Java classes, so we have chosen to begin this lab by modeling a very simple, everyday object: a door.

Write the code to create a class that models a door object. Don’t worry about the internal details just yet. Just give the class a name and an empty body for the moment. We will add more to the class shortly.

|  |
| --- |
| Public static class Door{    }    public static void main(String [] args)  {    } |

7.2) (2 pts.) When modeling an object as a class, we also need to describe the properties it possesses. An everyday object that we wish to model always has one or more properties that describe it. For instance a door object might have a name like “Front” or “Side” to distinguish it from other doors. Another property that could describe a door is its state: “open” or “closed”. Properties of objects are described in code by using nouns like “state” or “name” to create instance variables that hold values.

Add instance variables to your Door class for the name of the door and its state. Experience has shown that we almost always want to limit the visibility of instance variables inside the same class, so make the access modifiers of state and name private. And because the state and name properties have values like “open” or “front”, let the instance variables you create be of type String.

|  |
| --- |
| Public static class Door{  private String name;  private String state;  } |

7.3) (2 pts.) Objects also have operations which can be invoked on the object and which may change the object in some way. For instance, the operations for a door object could be “open” and “close”. An operation on an object corresponds to a Java method and is described in code by using a verb like “open” or “close”. Invoking a method may change the value of an instance variable. For example, invoking close() would change the value of the state variable from “open” to “closed”.

Declare methods for open and close. Because we usually want to allow free access to the methods a class contains, make the access modifier for each method public.

|  |
| --- |
| public static class Door{  private String name;  private String state;    public void close() {state = "Closed";}  public void open() {state = "Open";}  } |

7.4) (2 pts.) Now that we have a Door class, we would like to be able to create some Door objects. Java constructors are components of a class whose purpose is to create objects of the given class and to initialize the object’s instance variables. Java provides a constructor with no arguments (the “default” constructor) for every class and the Door class is no exception. Unfortunately, the default constructor that Java provides initializes the state and name variables to null. This is unacceptable.

Add a constructor for the Door class that receives two arguments: the name of the door and its initial state. Because we want to use the constructor outside of the class, make the access modifier for the constructor public.

|  |
| --- |
| public static class Door{  private String name;  private String state;    public void Door(String input\_name, String input\_state)  {  name = input\_name;  state = input\_state;  }  public void close() {state = "Closed";}  public void open() {state = "Open";}  } |

7.5) (2 pts.) It is often convenient to have accessor methods that operate on a single instance variable. Here is an accessor method for the name variable:

public String getName()

{

return name;

}

The word String in front of getName() indicates that the method returns a String when it is invoked. The body is simple and just returns the value of name.

Add this method to your class and write a similar accessor method for the instance variable state.

|  |
| --- |
| public static class Door{  private String name;  private String state;    public void Door(String input\_name, String input\_state)  {  name = input\_name;  state = input\_state;  }    public String getName(){return name;}  public String getState(){return state;}  public void close() {state = "Closed";}  public void open() {state = "Open";}  } |

7.6) (2 pts.) Many instance variables in a class will have corresponding mutator methods that allow you to change the value of the variable. Here is a mutator method for the name variable:

public void setName(String newName)

{

name = newName;

}

The word void in front of setName() indicates that the method does not return a value when it is invoked. The body is simple and copies the value of the parameter variable newName to instance variable name.

Add this method to the class and write a similar mutator method for the instance variable state.

|  |
| --- |
| public static class Door{  private String name;  private String state;    public void Door(String input\_name, String input\_state)  {  name = input\_name;  state = input\_state;  }    public String getName(){return name;}  public String getState(){return state;}  public void setName(String newName){name = newName;}  public void setState(String newState){state = newState;}  public void close() {state = "Closed";}  public void open() {state = "Open";}  } |

7.7) (2 pts.) If you are using BlueJ, use the interactive features of the IDE to create a Door object. Verify that it has the correct values for name and state. Invoke the setState mutator method on the Door object and verify that the value of the object’s state was changed. Repeat the process for the setName mutator. Unit test the code you have written until you are convinced it is correct.

If you are not using BlueJ, compile and run the code below:

/\*\*

A class to test the Door class.

\*/

public class DoorTester

{

/\*\*

Tests the methods of the Door class

@param args not used

\*/

public static void main(String[] args)

{

Door frontDoor = new Door("Front", "open");

System.out.println("The front door is " + frontDoor.getState());

System.out.println("Expected: open");

Door backDoor = new Door("Back", "closed");

System.out.println("The back door is " + backDoor.getState());

System.out.println("Expected: closed");

// Use the mutator to change the state variable

backDoor.setState("open");

System.out.println("The back door is " + backDoor.getState());

System.out.println("Expected: open");

// Add code to test the setName mutator here

}

}

Create a third Door object called “sideDoor” with the name property “Side” and an initial state of “closed”. Verify that the object was properly created. Use the mutator to change the state of object sideDoor to “open”. Verify that the mutator is working.

|  |
| --- |
| Door sideDoor = new Door("Side", "closed");  System.out.println("The side door is " + sideDoor.getState());  System.out.println("Expected: closed");  sideDoor.setState("open");  System.out.println("The side door is " + sideDoor.getState());  System.out.println("Expected: open");  **OUTPUT:**  **The side door is closed**  **Expected: closed**  **The side door is open**  **Expected: open** |

7.8) (2 pts.) Consider the variable state in the class Door and the variable newState in the mutator for state. What kind of variable is state? What kind of variable is newState? When do these variables exist?

**Both *state* and *newState* are of type String, the variable state exists as soon as a new instance of class *Door* is created and will continue to exist as long as the program is running and the instance of class *Door* has not been deleted. *newState* only exists from the time *setState()* is called until *setState()* finished being executed upon which it is deleted from memory**

7.9) (2 pts.) Consider the line below which was taken from the main method in Lab 8.1.7 above.

backDoor.setState("open");

What is the implicit parameter that is passed by this method call? What is the explicit parameter?

**The implicit parameter is the *backdoor* as it is the parameter that *setState()* relies on, since you cannot call *setState()* without first creating an object for it to point to *(backdoor)*. The explicit parameter is *“open”.***

8.1) (2 pts.) In this lab, you will implement a *vending machine* that holds cans of soda. To buy a can of soda, the customer needs to insert a token into the machine. When the token is inserted, a can drops from the can reservoir into the product delivery slot. The vending machine can be filled with more cans. The goal is to determine how many cans and tokens are in the machine at any given time.

What methods would you supply for a VendingMachine class? Describe them informally.

**First I would create two variables of type int, one for the amount of cans in the machine, and 1 for the amount of tokens. I would then make the default constructors and get/set methods for my variables as well as one method to display both variables .Next I would have a method for dispensing a can, which would decrement the cans variable and increment the tokens. I would also have 2 refill methods, one having no parameter that increments cans by 1, the other an overloaded version that accepts an int and increases can amount by that much.**

8.2) (2 pts.) Now translate those informal descriptions into Java method headers, such as

public void fillUp(int cans)

Give the names, parameter variables, and return types of the methods. Do not implement them yet.

|  |
| --- |
| Public void fillUp()  Public void fillUp(int newCans)  Public void insertToken()  Public int getTokenCount()  Public int getCanCount()  Public void setTokens(int tokenAmount)  Public void setCans(int canAmount)  Public void getCurrent()  Public void VendingMachine()  Public void VendingMachine(int canAmount, int tokenAmount) |

8.3) (2 pts.) What instance variables do the methods need to do their work? *Hint:* You need to track the number of cans and tokens. Declare them with their type and access modifier.

|  |
| --- |
| Private int cans  Private int tokens |

8.4) (2 pts.) Consider what happens when a user inserts a token into the vending machine. The number of tokens is increased, and the number of cans is decreased. Implement a method:

public void insertToken()

{

// Instructions for updating the token and can counts

}

You need to use the instance variables that you defined in the previous step.

Be sure to check that the number of cans is greater than zero before decreasing the can count and increasing the token count.

|  |
| --- |
| public void insertToken()  {  if(cans > 0)  {  --cans;  ++tokens;  }  else  System.out.println("ERROR: no more cans");  } |

8.5) (2 pts.) Now implement a method fillUp(int cans) to add more cans to the machine. Simply add the number of new cans to the can count.

|  |
| --- |
| public void fillUp() { cans++; }  public void fillUp(int numCans) { cans += numCans;} |

8.6) (2 pts.) Next, implement two methods, getCanCount and getTokenCount, that return the current values of the can and token counts.

|  |
| --- |
| public int getCanCount() { return cans; }  public int getTokenCount() { return tokens; } |

8.7) (2 pts.) You have implemented all methods of the VendingMachine class.

Put them together into a class, like this:

public class VendingMachine

{

private *your first instance variable*

private *your second instance variable*

public *your first method*

public *your second method*

. . .

}

|  |
| --- |
| public static class VendingMachine  {  private int tokens;  private int cans;    public void insertToken()  {  if(cans > 0)  {  --cans;  ++tokens;  }  else  System.out.println("ERROR: no more cans");  }  public void fillUp() { cans++; }  public void fillUp(int numCans) { cans += numCans;}  public int getCanCount() { return cans; }  public int getTokenCount() { return tokens; }  } |

8.8) (2 pts.) Now complete the following tester program so that it exercises all of the methods of your class.

public class VendingMachineTester

{

public static void main(String[] args)

{

VendingMachine machine = new VendingMachine();

machine.fillUp(10); // Fill up with ten cans

machine.insertToken();

machine.insertToken();

System.out.print("Token count: ");

System.out.println(machine.getTokenCount());

System.out.println("Expected: . . .");

System.out.print("Can count: ");

System.out.println(machine.getCanCount());

System.out.println("Expected: . . .");

}

}

|  |
| --- |
| **OUTPUT:**  **Token count: 2**  **Expected: . . .**  **Can count: 8**  **Expected: . . .** |

8.9) (2 pts.) So far, the VendingMachine class does not have any constructors. Instances of a class with no constructor are always constructed with all instance variables set to zero (or null if they are object references). It is always a good idea to provide an explicit constructor.

Provide two constructors for the VendingMachine class:

a) A constructor with no arguments that initializes the vending machine with 10 soda cans

b) A constructor, VendingMachine(int cans), that initializes the vending machine with the given number of cans

Both constructors should initialize the token count to 0.

|  |
| --- |
| public static class VendingMachine  {  private int tokens;  private int cans;    public VendingMachine()  {  tokens = 0;  cans = 10;  }  public VendingMachine(int numCans)  {  tokens = 0;  cans = numCans;  }  public void insertToken()  {  if(cans > 0)  {  --cans;  ++tokens;  }  else  System.out.println("ERROR: no more cans");  }  public void fillUp() { cans++; }  public void fillUp(int numCans) { cans += numCans;}  public int getCanCount() { return cans; }  public int getTokenCount() { return tokens; }  } |

9.1) (3 pts.) Consider the following task: You are on vacation and want to send postcards to your friends. A typical postcard might look like this:

Dear Sue: I am having a great time on

the island of Java. The weather

is great. Wish you were here!

Love,

Janice

You decide to write a computer program that sends postcards to various friends, each of them with the same message, except that the first name is substituted to match each recipient.

What "black box" (class that will be used to build objects of its type) can you identify that will allow you to send your greeting?

9.2) (3 pts.) We want to be able to write a program that will use our Postcard class to send postcards with the same message to different recipients.

The following class implements a Postcard. Notice that we do not set the recipient in the constructor because we want to be able to change the recipient, and keep the same message and sender. What method would you add to support this functionality? Implement the method.

public class Postcard

{

private String message;

private String sender;

private String recipient;

public Postcard(String aSender, String aMessage)

{

message = aMessage;

sender = aSender;

recipient = "";

}

// Your method here

|  |
| --- |
| Public void setRecipient(String aRecipient)  { recipient = ("Dear " +aRecipient + ": "); } |

}

9.3) (3 pts.) Add a print method to the Postcard class to display the contents of the postcard on the screen. Use the instance variables message, sender, and recipient defined in the last step when you implement the method.

|  |
| --- |
| public void print()  {  System.out.println(recipient + message + "Love,\n" + sender);  } |

9.4) (3 pts.) Try out your class with the following code:

public class PostcardPrinter

{

public static void main(String[] args)

{

String text = "I am having a great time on\nthe island of Java. Weather\nis great. Wish you were here!";

Postcard postcard = new Postcard("Janice", text);

postcard.setRecipient("Sue");

postcard.print();

postcard.setRecipient("Tim");

postcard.print();

}

}

What is the output of your program?

|  |
| --- |
| **OUTPUT:**  **Dear Sue: I am having a great time on**  **the island of Java. Weather**  **is great. Wish you were here!**  **Love,**  **Janice**  **Dear Tim: I am having a great time on**  **the island of Java. Weather**  **is great. Wish you were here!**  **Love,**  **Janice** |

10.1) (5 pts.) Using the Person and PersonRunner classes below, execute the main method in PersonRunner. Why does the program end abnormally? Comment out the first two lines and run the program again. Why does the reference to p3 cause Frank’s name to be printed when we set Frank’s name with the p2 reference?

public class PersonRunner

{

public static void main(String[] args)

{

Person p1 = null;

p1.speak();

Person p2 = new Person("Beth");

p2.speak();

Person p3 = p2;

p2.speak();

p2.setName("Frank");

p3.speak();

}

}

public class Person

{

private String name;

public Person(String name)

{

this.name = name;

}

public void setName(String name)

{

this.name = name;

}

public String getName()

{

return name;

}

public void speak()

{

System.out.println("My name is " + name);

}

}

**The program ends abnormally because went *p1* initialized it is set to NULL, meaning it is pointing to a space in memory that contains the machine equivalent of nothing. Trying to print “nothing” causes compiler errors. As for *p3* displaying the name frank when logically it should still be set to beth is because *Person p3 = p2* does not create a new *person* with *p2*’s properties, it instead creates a pointer to *p2*, they will always contain the “same” thing.**

10.2) (5 pts.) Modify the Person and PersonRunner classes from Lab 10.1. Add a static int variable count in the Person class. Increment the count field each time a Person object is instantiated. Add a static method in Person called printCount() that prints a message that indicates the number of Person objects that have been created. Modify the main program so that it creates four Person objects and then invokes printCount().

|  |
| --- |
| public static void main(String[] args)  {  Person p1 = new Person("Test1");  p1.printCount();  Person p2 = new Person("Test2");  p1.printCount();  Person p3 = new Person("Test3");  p1.printCount();  Person p4 = new Person("Test4");  p1.printCount();  }  public static class Person  {  private String name;  private static int count = 0;    public Person()  {  name = null;  count++;  }  public Person(String name)  {  this.name = name;  count++;  }  public void setName(String name)  {  this.name = name;  }  public String getName()  {  return name;  }  public void speak()  {  System.out.println("My name is " + name);  }  public void printCount()  {  System.out.println("Current count: " + count);  }  }  **OUTPUT:**  **Current count: 1**  **Current count: 2**  **Current count: 3**  **Current count: 4** |