Lab Questions
Big Java, Late Objects / Java for Everyone, 2e
Chapter Number: 5 Methods

1) 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.

2) 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...");
   }
}
```

3) 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;
   }
}
```

4.1) 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 `*****=====*****`.

4.2) 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, "*")`.

4.3) 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, "*")`.

4.4) 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.

4.5) 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, "*")`.

5.1) 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));
   }

   // Your code goes here
}
```

5.2. 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 5.5.1. 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));
   }
   // Put your code here
}
```

6) 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 a 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"`.

7) Recursion is an elegant, magical way to solve problems. Problems that are difficult to solve with an iterative technique are sometimes quite simple when solved with a recursive technique. Good programmers understand how to program in both styles. The basic idea is to take a "large" problem and imagine how a solution to a "smaller" version of the problem could be used to solve the original problem.

Consider the problem of reversing the letters in a string. We will need a couple of `String` methods. Assume `String s = "abcde"`.

a) `s.charAt(0)` returns the character `"a"`.
b) `s.substring(1)` returns the string `"bcde"`

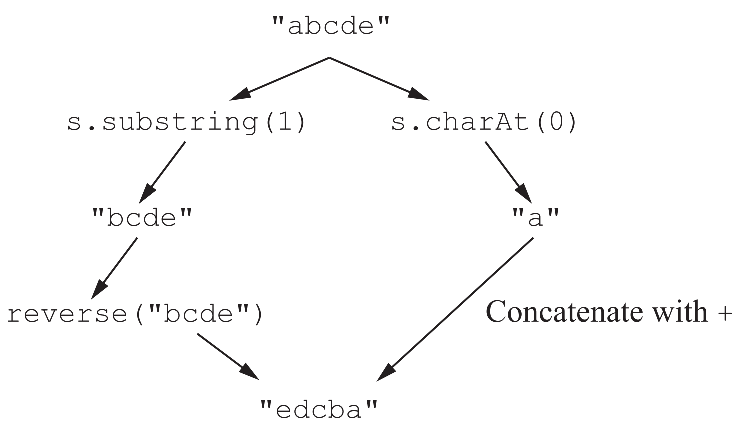
Let's plan a method for reversing the string that uses these methods. If the problem at hand is reversing `s`, we first need to think of a smaller related problem. Can you imagine one? Suppose we consider the problem of reversing the string "bcde"? Could the solution to this problem help us solve the original problem?

Here is a diagram of the solution:

Original problem: Reverse `"abcde"`

Shrink the problem: Reverse `"bcde"`

Use this solution to solve the original problem.



One of the steps in the above process involves calling `reverse("bcde")`. The interesting thing about recursive methods is that you can solve the smaller problems by calling the method you are writing. Keep in mind you don't have to show how to solve all the smaller problems, you just have to show how a solution to a smaller problem can be used to solve the larger problem.

The process of "shrinking" the original problem can't continue forever, so we also have to specify the solutions to the smallest problems we encounter. In the example above, if the string `s` has a single character, we can simply return it as the solution (you can reverse a string with only one character). You can test for this by examining whether `s.length() == 1`. In a recursive solution, you have to test for the smallest cases first.

Write a recursive method called `reverse` that is passed a `String` and that returns a `String` with the characters of the original string reversed. Use the following `main` method for testing your method.

```
public class Reverse
{
   public static void main(String[] args)
   {
      String word = "abcdefg";
      System.out.println("Word:  " + word );
      System.out.println("Word reversed: " + reverse(word));
   }
```

```
      // Put your code here
}
```