SP2 — Lab sheet 2

2014

Based upon exercises from Java for Everyone, 2e, Chapters 1 through 5.
Most of these you have probably encountered during Software and Programming I, although the recursion questions should be "new".

---

1. There is a famous story about a primary school teacher who wanted to occupy his students' time by making the children compute the sum of $1+2+3+\cdots+100$ by hand. As the story goes, the teacher was astounded when one of the children immediately produced the correct answer: 5050. The student, a child prodigy, was *Carl Gauss*, who grew up to be one of the most famous mathematicians of the eighteenth century. Repeat Gauss's remarkable calculation by writing a loop that will compute and print the above sum. After you have the program working, rewrite it so you can compute $1+2+\cdots+n$ where n is any positive integer.

2. Java provides three types of loops: `while`, `for`, and `do` (also called `do-while`). Theoretically, they are interchangeable — any program you write with one kind of loop could be rewritten using any of the other types of loops. As a practical matter, though, it is often the case that choosing the right kind of loop will make your code easier to produce, debug, and read. It takes time and experience to learn to make the best loop choice, so this is an exercise to give you some of that experience.

   Rewrite Question 1 using a `for` loop. Repeat the exercise again but this time use a `do while` loop. Which form of loop seems to work best? Why?

3. One of the oldest numerical algorithms was described by the Greek mathematician, Euclid, in 300 B.C. That algorithm is described in Book VII of Euclid's multi-volume work Elements. It is a simple but very effective algorithm that computes the greatest common divisor of two given integers.

   For instance, given integers 24 and 18, the greatest common divisor is 6, because 6 is the largest integer that divides evenly into both 24 and 18. We will denote the greatest common divisor of x and y as $gcd(x, y)$. The algorithm is based on the clever idea that the $gcd(x, y) = gcd(xy, y) if x >= y$. The algorithm consists of a series of steps (loop iterations) where the *larger* integer is replaced by the difference of the larger and smaller integer.

In the example below, we compute $gcd(72, 54)$ and list each loop iteration computation on a separate line. The whole process stops when one of the integers becomes zero. When this happens, the greatest common divisor is the non-zero integer.

```
gcd(72, 54) = gcd(72  54, 54) = gcd(18, 54)

gcd(18, 54) = gcd(18, 54  18) = gcd(18, 36)

gcd(18, 36) = gcd(18, 36  18) = gcd(18, 18)

gcd(18, 18) = gcd(18  18, 18) = gcd(0, 18) = 18
```

To summarize:

1. Create a loop, and subtract the smaller integer from the larger one (if the integers are equal you may choose either one as the *larger*) during each iteration.
2. Replace the larger integer with the computed difference.
3. Continue looping until one of the integers becomes zero.
4. Print out the non-zero integer.

Use the code below to prompt the user for the two integers.

```
import java.util.Scanner;

public class GCD {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    System.out.println("Enter the first integer: ");
    int x = in.nextInt();
    System.out.println("x = " + x);
    System.out.println("Enter the second integer: ");
    int y = in.nextInt();
    System.out.println("y = " + y);
    // Your gcd computation code goes here
    in.close();
  }
}
```

4. One loop type might be better suited than another to a particular purpose. The following usages are idiomatic:

   `for` Known number of iterations

   `while` Unknown number of iterations

   `do` At least one iteration

   Convert the following `while` loop to a `do` loop.

```
import java.util.Scanner;

public class PrintSum {
  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    int sum = 0;
    int n = 1;

    while (n != 0) {
      System.out.print("Please enter a number, 0 to quit: ");
      n = in.nextInt();
      if (n != 0) {
        sum = sum + n;
        System.out.println("Sum = " + sum);
      }
    }
    in.close();
  }
}
```

Is the `do` loop an improvement over the `while` loop? Why or why not?

5. 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.

6. Run the following code. The `sum` method in this program violates an accepted design principle 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. 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;
  }
}
```

7. 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

   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:

   1. If the digit is in an odd-numbered position, simply return the digit,
   2. 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));
  }

  int decode(int number, boolean position) {
    // Your code goes here
  }

}
```

8. 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"`.

9. 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.

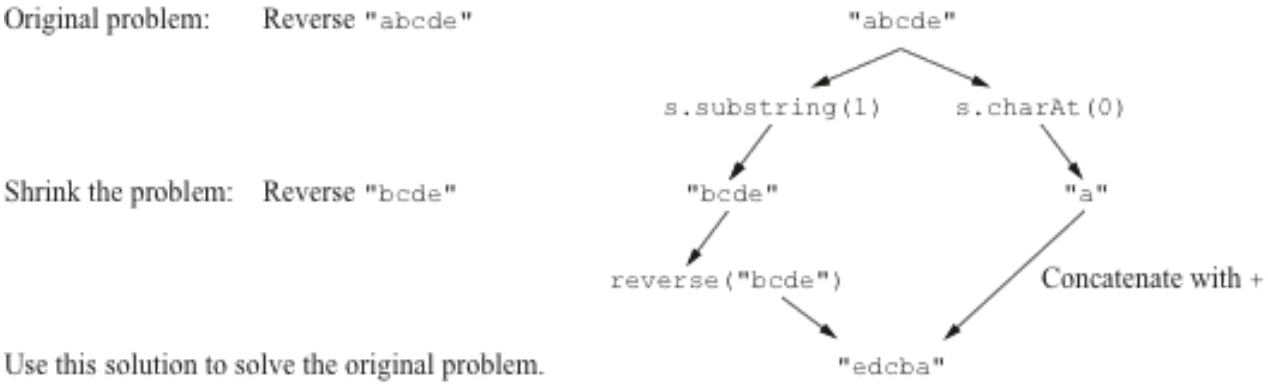
   Consider the problem of reversing the letters in a string. We will need a couple of `String` methods. Assume `String s = "abcde"`.

   (a) `s.charAt(0)` returns the character `"a"`

   (b) `s.substring(1)` returns the string `"bcde"`

   Lets 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?



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 dont 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 cant 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
}
```