Chapter 1: Introduction to Java Programming

Exercise 1.1: Stewie

Write a complete Java program in a class named Stewie that prints the following output:

```
//////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
```

```
public class Stewie {
   public static void main(String[] args) {
      System.out.println("//////////////////////");
      System.out.println("|| Victory is mine! ||");
      System.out.print("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\");
   }
}
```

Exercise 1.2: Spikey

Write a complete Java program in a class named Spikey that prints the following output:

```
  \/
 \\//
\\\///
///\\\
 //\\
  /\
```

```
public class Spikey {
   public static void main(String[] args) {
      System.out.println("  \\/");
      System.out.println(" \\\\//");
      System.out.println("\\\\\\///");
      System.out.println("///\\\\\\");
      System.out.println(" //\\\\");
      System.out.print("  /\\");
   }
}
```

Exercise 1.3: WellFormed

Write a complete Java program in a class named WellFormed that prints the following output:

```
A well-formed Java program has
```

```
a main method with { and }
braces.

A System.out.println statement
has ( and ) and usually a
String that starts and ends
with a " character.
(But we type \" instead!)
```

```
public class WellFormed {
    public static void main(String[] args) {
        System.out.println("A well-formed Java program has");
        System.out.println("a main method with { and }");
        System.out.println("braces.\n");
        System.out.println("A System.out.println statement");
        System.out.println("has ( and ) and usually a");
        System.out.println("String that starts and ends");
        System.out.println("with a \" character.");
        System.out.println("(But we type \\\" instead!)");
    }
}
```

Exercise 1.4: Difference

Write a complete Java program in a class named Difference that prints the following output:

```
What is the difference between
a ' and a "?  Or between a " and a \"?

One is what we see when we're typing our program.
The other is what appears on the "console."
```

```
public class Difference {
    public static void main(String[] args) {
        System.out.println("What is the difference between");
        System.out.println("a ' and a \"?  Or between a \" and a \\\"?\n");
        System.out.println("One is what we see when we're typing our program.");
        System.out.println("The other is what appears on the \"console.\"");
    }
}
```

Exercise 1.5: MuchBetter

Write a complete Java program in a class named MuchBetter that prints the following output:

```
A "quoted" String is
'much' better if you learn
the rules of "escape sequences."
Also, "" represents an empty String.
Don't forget: use \" instead of " !
" is not the same as "
```

```
public class MuchBetter {
    public static void main(String[] args) {
        System.out.println("A \"quoted\" String is");
        System.out.println("'much' better if you learn");
        System.out.println("the rules of \"escape sequences.\"");
        System.out.println("Also, \"\" represents an empty String.");
        System.out.println("Don't forget: use \\\" instead of \" !");
        System.out.println("\" is not the same as \"");
    }
}
```

Exercise 1.6: Meta

Write a complete Java program called Meta whose output is the text that would be the source code of a Java program named Hello that prints "Hello, world!" as its output:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Your program must produce *exactly* the output shown in order to pass (using exactly four spaces for each increment of indentation in the output).

```
public class Meta {
    public static void main(String[] args) {
        System.out.println("public class Hello {");
        System.out.println("    public static void main(String[] args) {");
        System.out.println("        System.out.println(\"Hello, world!\");");
        System.out.println("    }");
        System.out.println("}");
    }
}
```

Exercise 1.7: Mantra

Write a complete Java program in a class named Mantra that produces the following output. Remove its redundancy by adding a method.

```
There's one thing every coder must understand:
The System.out.println command.

There's one thing every coder must understand:
The System.out.println command.
```

```
public class Mantra {
   public static void main(String[] args) {
      printParagraph();
      System.out.println();
      printParagraph();
   }

   public static void printParagraph() {
      System.out.println("There's one thing every coder must understand:");
      System.out.println("The System.out.println command.");
   }
}
```

Exercise 1.8: Stewie2

Write a complete Java program in a class named Stewie2 that prints the following output. Use at least one static method besides main to remove redundancy.

```
//////////////////////
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
|| Victory is mine! ||
\\\\\\\\\\\\\\\\\\\\\\
```

```
public class Stewie2 {
   public static void main(String[] args) {
      printForward();
      for(int i = 0; i < 5; i++)
         printVictory();
```

```
    }

    public static void printForward() {
        System.out.println("//////////////////////");
    }

    public static void printVictory() {
        System.out.println("|| Victory is mine! ||");
        System.out.println("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\");
    }
}
```

Exercise 1.9: Egg

Write a complete Java program in a class named Egg that displays the following output:

```
  _______
 /       \
/         \
-"-'-"-'-"-
\         /
 \_______/
```

```
public class Egg {
    public static void main(String[] args) {
        System.out.println("  _______");
        System.out.println(" /       \\");
        System.out.println("/         \\");
        System.out.println("-\"-'-\"-'-\"-");
        System.out.println("\\         /");
        System.out.println(" \\_______/");
    }
}
```

Exercise 1.10: Egg2

Write a complete Java program in a class named Egg2 that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
  _______
 /       \
/         \
\         /
 \_______/
-"-'-"-'-"-
  _______
 /       \
/         \
\         /
 \_______/
-"-'-"-'-"-
\         /
 \_______/
  _______
 /       \
/         \
-"-'-"-'-"-
\         /
 \_______/
```

```
public class Egg2 {
    public static void main(String[] args) {
        printTop();
        printBottom();
        printMiddle();
        printTop();
        printBottom();
        printMiddle();
        printBottom();
        printTop();
        printMiddle();
        printBottom();
    }

    public static void printTop() {
        System.out.println("  _______");
        System.out.println(" /       \\");
        System.out.println("/         \\");
    }

    public static void printMiddle() {
        System.out.println("-\"-'-\"-'-\"-");
    }

    public static void printBottom() {
        System.out.println("\\         /");
        System.out.println(" \\_______/");
    }
}
```

Exercise 1.11: TwoRockets

Write a complete Java program in a class named TwoRockets that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

Note that there are two rocket ships next to each other. What redundancy can you eliminate using static methods? What redundancy cannot be eliminated?

```
   /\       /\
  /  \     /  \
 /    \   /    \
+------+ +------+
|      | |      |
|      | |      |
+------+ +------+
|United| |United|
|States| |States|
+------+ +------+
|      | |      |
|      | |      |
+------+ +------+
   /\       /\
  /  \     /  \
 /    \   /    \
```

```
public class TwoRockets {
   public static void main(String[] args) {
      printTriangle();
      printSquare();
      printUSA();
      printSquare();
      printTriangle();
   }

   public static void printTriangle() {
      System.out.println("   /\\       /\\");
      System.out.println("  /  \\     /  \\");
      System.out.println(" /    \\   /    \\");
   }

   public static void printSquare() {
      System.out.println("+------+ +------+");
      System.out.println("|      | |      |");
      System.out.println("|      | |      |");
      System.out.println("+------+ +------+");
   }
```

```
    public static void printUSA() {
        System.out.println("|United| |United|");
        System.out.println("|States| |States|");
    }
}
```

Exercise 1.12: FightSong

Write a complete Java program in a class named FightSong that generates the following three figures of output. Use static methods to show structure and eliminate redundancy in your solution.

In particular, make sure that main contains no System.out.println statements, that any System.out.println statement(s) repeated are captured in a method that is called just once, and that the same System.out.println statement never appears in two places in the code.

```
Go, team, go!
You can do it.

Go, team, go!
You can do it.
You're the best,
In the West.
Go, team, go!
You can do it.

Go, team, go!
You can do it.
You're the best,
In the West.
Go, team, go!
You can do it.

Go, team, go!
You can do it.
```

```
public class FightSong {
    public static void main(String[] args) {
        printGo();
        printNewLine();
        printParagraph();
        printNewLine();
        printParagraph();
        printNewLine();
        printGo();
    }

    public static void printGo() {
        System.out.println("Go, team, go!");
        System.out.println("You can do it.");
    }

    public static void printWest() {
        System.out.println("You're the best,");
        System.out.println("In the West.");
    }

    public static void printNewLine() {
        System.out.println();
    }

    public static void printParagraph() {
        printGo();
        printWest();
        printGo();
    }
}
```

Exercise 1.13: StarFigures

Write a complete Java program in a class named StarFigures that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
*****
*****
 * *
  *
 * *

*****
*****
 * *
  *
 * *
*****
*****

  *
  *
  *
*****
*****
 * *
  *
 * *
```

```
public class StarFigures {
   public static void main(String[] args) {
      printLines();
      printX();
      System.out.println();
      printLines();
      printX();
      printLines();
      System.out.println();
      printThreeStars();
      printLines();
      printX();
   }

   public static void printLines() {
      System.out.println("*****");
      System.out.println("*****");
   }

   public static void printX() {
      System.out.println(" * *");
      System.out.println("  *");
      System.out.println(" * *");
   }

   public static void printThreeStars() {
      System.out.println("  *");
      System.out.println("  *");
      System.out.println("  *");
   }
}
```

Exercise 1.14: Lanterns

Write a complete Java program in a class named Lanterns that generates the following three figures of output. Use static methods to show structure and eliminate redundancy in your solution.

In particular, make sure that main contains no System.out.println statements except for blank lines, that any System.out.println statement(s) repeated are captured in a method that is called just once, and that the same System.out.println statement never appears in two places in the code.

```
    *****
  *********
*************

    *****
  *********
*************
* | | | | | *
*************
    *****
  *********
*************

    *****
  *********
*************
    *****
* | | | | | *
* | | | | | *
    *****
    *****
```

```
public class Lanterns {
    public static void main(String[] args) {
        printTwoTriangles();
        printBars();
        printLine();
        printTwoTriangles();
        printFiveStars();
        printBars();
        printBars();
        printFiveStars();
        printFiveStars();
    }

    public static void printTwoTriangles() {
        printTriangle();
        System.out.println();
        printTriangle();
    }
    public static void printTriangle() {
        System.out.println("    *****");
        System.out.println("  *********");
        System.out.println("*************");
    }

    public static void printBars() {
        System.out.println("* | | | | | *");
    }
                                                          public static void printLine() {
        System.out.println("*************");
    }

    public static void printFiveStars() {
        System.out.println("    *****");
    }
}
```

Exercise 1.15: EggShop

Write a complete Java program in a class named EggStop that generates the following output. Use static methods to show structure and eliminate redundancy in your solution.

```
  _______
 /       \
/         \
\         /
 \_______/

  _______
 /       \
/         \
\         /
 \_______/
+--------+

  _______
 /       \
/         \
|   STOP  |
\         /
 \_______/
+--------+
public class EggStop {
    public static void main(String[] args) {
        printEgg();
        System.out.println();
        printEgg();
        printLine();
        System.out.println();
        printTop();
        printStop();
        printBottom();
        printLine();
    }

    public static void printEgg() {
        printTop();
        printBottom();
    }

    public static void printLine() {
        System.out.println("+--------+");
    }

    public static void printTop() {
        System.out.println("  ______");
        System.out.println(" /      \\");
        System.out.println("/        \\");
    }

    public static void printBottom() {
        System.out.println("\\        /");
        System.out.println(" \\______/");
    }

    public static void printStop() {
        System.out.println("|  STOP  |");
    }
}
```

Exercise 1.16: Shining

Write a program in a class named Shining that prints the following line of output 1000 times:

```
All work and no play makes Jack a dull boy.
```

You should not write a program whose source code is 1000 lines long; use methods to shorten the program. What is the shortest program you can write that will produce the 1000 lines of output, using only the material from Chapter 1 (println, methods, etc.)?

```
public class Shining {
   public static void main(String[] args) {
      printFiveHundred();
      printFiveHundred();
   }

   public static void printFiveHundred() {
      printOneHundred();
      printOneHundred();
      printOneHundred();
      printOneHundred();
      printOneHundred();
   }

   public static void printOneHundred() {
      printFifty();
      printFifty();
   }

   public static void printFifty() {
      printTen();
      printTen();
      printTen();
      printTen();
      printTen();
   }

   public static void printTen() {
      printFive();
      printFive();
   }

   public static void printFive() {
      System.out.println("All work and no play makes Jack a dull boy.");
      System.out.println("All work and no play makes Jack a dull boy.");
      System.out.println("All work and no play makes Jack a dull boy.");
      System.out.println("All work and no play makes Jack a dull boy.");
      System.out.println("All work and no play makes Jack a dull boy.");
   }
}
```

Chapter 2: Primitive Data and Definite Loops

Exercise 2.1: displacement

In physics, a common useful equation for finding the position *s* of a body in linear motion at a given time *t*, based on its initial position s_0, initial velocity v_0, and rate of acceleration *a*, is the following:

$$s = s_0 + v_0 t + \frac{1}{2} at^2$$

Write code to declare variables for s_0 with a value of 12.0, v_0 with a value of 3.5, *a* with a value of 9.8, and *t* with a value of 10, and then write the code to compute *s* on the basis of these values. At the end of your code, print the value of your variable *s* to the console.

```
double s0 = 12.0;
double v0 = 3.5;
double a = 9.8;
double t = 10.0;

double s = s0 + v0 * t + 0.5 * a * t * t;
System.out.println(s);
```

Exercise 2.2: loopSquare

Write a for loop that produces the following output:

```
1 4 9 16 25 36 49 64 81 100
```

For added challenge, try to modify your code so that it does not need to use the * multiplication operator. (It can be done! Hint: Look at the differences between adjacent numbers.)

```
for(int i = 1, inc = 3; i <= 100; inc += 2) {
   System.out.print(i + " ");
   i += inc;
}
```

Exercise 2.3: Fibonacci

The Fibonacci numbers are a sequence of integers in which the first two elements are 1, and each following element is the sum of the two preceding elements. The mathematical definition of each k^{th} Fibonacci number is the following:

F(k):

k > 2 : F(k-1) + F(k-2)

k <= 2 : 1

The first 12 Fibonacci numbers are:

```
1 1 2 3 5 8 13 21 34 55 89 144
```

Write a piece of code that uses a for loop to compute and print the first 12 Fibonacci numbers. (You may include other code, such as declaring variables before the loop, if you like.)

```
int f0 = 1;
int f1 = 1;

System.out.print(f0 + " ");
System.out.print(f1 + " ");

for(int fk = f1 + f0; fk <= 144; fk = f1 + f0) {
   System.out.print(fk + " ");
   f0 = f1;
   f1 = fk;
}
```

Exercise 2.4: startSquare

Write for loops to produce the following output:

```
*****
*****
*****
*****
```

```
for(int i = 0; i < 4; i++) {
   for(int j = 0; j < 5; j++)
      System.out.print("*");
   System.out.println();
}
```

Exercise 2.5: starTriangle

Write for loops to produce the following output:

```
*
```

```
**
***
****
*****
```

```
for(int i = 1; i <= 5; i++) {
    for(int j = 0; j < i; j++)
        System.out.print("*");
    System.out.println();
}
```

Exercise 2.6: numberTriangle

Write for loops to produce the following output:

```
1
22
333
4444
55555
666666
7777777
```

```
for(int i = 1; i < 8; i++) {
    for(int j = 0; j < i; j++)
        System.out.print(i);
    System.out.println();
}
```

Exercise 2.7: spacedNumbers

Write nested for loops to produce the following output:

```
    1
   2
  3
 4
5
```

```
for(int i = 1; i <= 5; i++) {
    for(int j = 5 - i; j > 0; j--) {
        System.out.print(" ");
    }
    System.out.println(i);
}
```

Exercise 2.8: spacesAndNumbers

Write nested for loops to produce the following output:

```
    1
   22
  333
 4444
55555
```

```
for(int i = 1; i <= 5; i++) {
    for(int j = 5 - i; j > 0; j--)
        System.out.print(" ");

    for(int j = 0; j < i; j++)
        System.out.print(i);

    System.out.println();
}
```

Exercise 2.9: waveNumbers40

Write for loops to produce the following output, with each line 40 characters wide:

```
----------------------------------------
_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-
1122334455667788990011223344556677889900
----------------------------------------
```

```
for(int i = 0; i < 40; i++)
    System.out.print("-");

System.out.println();

for(int i = 0; i < 10; i++)
    System.out.print("_-^-");

System.out.println();

for(int j = 0; j < 2; j++) {
    for(int i = 1; i <= 10; i++) {
        System.out.print(i%10);
        System.out.print(i%10);
    }
}

System.out.println();

for(int i = 0; i < 40; i++)
    System.out.print("-");
```

Exercise 2.10: numberOutput60

It's common to print a rotating, increasing list of single-digit numbers at the start of a program's output as a visual guide to number the columns of the output to follow. With this in mind, write nested for loops to produce the following output, with each line 60 characters wide:

```
         |         |         |         |         |         |
123456789012345678901234567890123456789012345678901234567890
```

```
for(int i = 0; i < 6; i++) {
    for(int j = 0; j < 9; j++)
        System.out.print(" ");
    System.out.print("|");
}

System.out.println();

for(int i = 1; i <= 60; i++)
    System.out.print(i%10);
```

Exercise 2.11: numbersOutputConstant

Modify your code from the previous exercise so that it could easily be modified to display a different range of numbers (instead of 1234567890) and a different number of repetitions of those numbers (instead of 60 total characters), with the vertical bars still matching up correctly. Write a complete class named NumbersOutput. Use two class constants instead of "magic numbers,", with one constant set to 6 for the number of repetitions, and the other set to 10 for the range of numbers. Put the for loop code in your class's main method.

For example, if your number-of-repetitions constant is set to 7 and your range constant is set to 5, the output should be the following:

```
    |    |    |    |    |    |    |
12340123401234012340123401234012340
```

```
public class NumbersOutput {
    public static final int NUM_REPETITIONS = 6;
    public static final int NUM_RANGE = 10;

    public static void main(String[] args) {
        for(int i = 0; i < NUM_REPETITIONS; i++) {
            for(int j = 0; j < NUM_RANGE - 1; j++)
                System.out.print(" ");

            System.out.print("|");
        }

        System.out.println();

        for(int i = 1; i <= NUM_REPETITIONS * NUM_RANGE; i++)
            System.out.print(i % NUM_RANGE);
    }
}
```

Exercise 2.12: nestedNumbers

Write nested for loops that produce the following output:

```
000111222333444555666777888999
000111222333444555666777888999
000111222333444555666777888999
```

```
for(int i = 0; i < 3; i++) {
    for(int j = 0; j <= 9; j++) {
        for(int k = 0; k < 3; k++)
            System.out.print(j);
    }
    System.out.println();
}
```

Exercise 2.13: nestedNumbers2

Modify your code from the previous problem to produce the following output:

```
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
```

```
for(int i = 0; i < 5; i++) {
    for(int j = 9; j >= 0; j--) {
        for(int k = 0; k < 5; k++)
            System.out.print(j);
    }
    System.out.println();
}
```

Exercise 2.14: nestedNumbers3

Modify your code from the previous problem to produce the following output:

```
999999999888888887777777666666555554444333221
999999999888888887777777666666555554444333221
999999999888888887777777666666555554444333221
999999999888888887777777666666555554444333221
```

```
for(int i = 0; i < 4; i++) {
    for(int j = 9; j >= 1; j--) {
        for(int k = 0; k < j; k++)
            System.out.print(j);
    }
    System.out.println();
}
```

Exercise 2.15: printDesign

Write a method called printDesign that produces the following output. Use nested for loops to capture the structure of the figure.

```
-----1-----
----333----
---55555---
--7777777--
-999999999-
```

```
public static void printDesign() {
    for(int i = 1; i <= 9; i+=2) {
        for(int j = 0; j < (11 - i) / 2; j++)
            System.out.print("-");

        for(int j = 0; j < i; j++)
            System.out.print(i);

        for(int j = 0; j < (11 - i) / 2; j++)
            System.out.print("-");

        System.out.println();
    }
}
```

Exercise 2.16: SlashFigure

Write a Java program in a class named SlashFigure that produces the following output. Use nested for loops to capture the structure of the figure. (If you previously solved Self-Check problems 34 and 35 in the book, you will have created a loop table that will be useful in solving this problem. Use it!)

```
!!!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!//
\\\\!!!!!!!!!!!!!!////
\\\\\\!!!!!!!!!!//////
\\\\\\\\!!!!!!////////
\\\\\\\\\\!!//////////
```

```
public class SlashFigure {
   public static void main(String[] args) {
      for(int i = 1; i <= 6; i++) {
         for(int j = 0; j < 2 * i - 2; j++)
            System.out.print("\\");

         for(int j = 0; j < -4 * i + 26; j++)
            System.out.print("!");

         for(int j = 0; j < 2 * i - 2; j++)
            System.out.print("/");

         System.out.println();
      }
   }
}
```

Exercise 2.16: SlashFigure2

Modify the SlashFigure program from the previous exercise to produce a new program SlashFigure2 that uses a global constant for the figure's height. The previous output used a constant height of 6. Here is the outputs for a constant height of 4 and 7 respectively: (If you previously solved Self-Check problems 34 and 35 in the book, you will have created a loop table that will be useful in solving this problem. Use it!)

size 4	**size 7**
!!!!!!!!!!!!!! \\!!!!!!!!!!//	!!!!!!!!!!!!!!!!!!!!!!!!!! \\!!!!!!!!!!!!!!!!!!!!!!//

\\\\!!!!!!//// \\\\\\!!//////	\\\\!!!!!!!!!!!!!!!!!!//// \\\\\\!!!!!!!!!!!!!!////// \\\\\\\\!!!!!!!!!!//////// \\\\\\\\\\!!!!!!////////// \\\\\\\\\\\\!!////////////

(You must solve this problem using only ONE public static final constant, not multiple constants; and its value must be used in the way described in this problem.)

```
public class SlashFigure2 {
   public static final int SIZE = 6;

   public static void main(String[] args) {
      for(int i = 1; i <= SIZE; i++) {
         for(int j = 0; j < 2 * i - 2; j++)
            System.out.print("\\");

         for(int j = 0; j < -4 * i + 4 * SIZE + 2; j++)
            System.out.print("!");

         for(int j = 0; j < 2 * i - 2; j++)
            System.out.print("/");

         System.out.println();
      }
   }
}
```

Exercise 2.18: pseudocodeWindow

Write a pseudocode algorithm that will produce the following figure as output:

```
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
```

(Since this is just pseudo-code, Practice-It is not able to verify that your solution is "correct". But you can submit an answer anyway to show that you completed this exercise. Any non-trivial answer will be accepted.)

```
public class Window {
   public static final int SIZE = 3;

   public static void main(String[] args) {
      printLine();
      System.out.println();
      printWindow();
      printLine();
      System.out.println();
      printWindow();
      printLine();
   }

   public static void printLine() {
      System.out.print("+");

      for(int i = 0; i < SIZE; i++)
         System.out.print("=");

      System.out.print("+");

      for(int i = 0; i < SIZE; i++)
         System.out.print("=");

      System.out.print("+");
   }

   public static void printWindow() {
      for(int i = 0; i < SIZE; i++) {
         System.out.print("|");

         for(int j = 0; j < SIZE; j++)
            System.out.print(" ");

         System.out.print("|");

         for(int j = 0; j < SIZE; j++)
            System.out.print(" ");

         System.out.print("|");
         System.out.println();
      }
```

```
    }
}
```

Exercise 2.19: Window

Write a Java program in a class named Window that produces the preceding figure as output. Use nested for loops to print the repeated parts of the figure. Once you get it to work, add one class constant to your program so that the size of the figure can be changed simply by changing that constant's value. The example output shown is at a constant size of 3, but if you change the constant, the figure should grow larger and wider proportionally.

```
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
```

(You must solve this problem using only ONE public static final constant, not multiple constants; and its value must be used in the way described in this problem.)

```
public class Window {
    public static final int SIZE = 3;

    public static void main(String[] args) {
        printLine();
        System.out.println();
        printWindow();
        printLine();
        System.out.println();
        printWindow();
        printLine();
    }

    public static void printLine() {
        System.out.print("+");

        for(int i = 0; i < SIZE; i++)
            System.out.print("=");

        System.out.print("+");
        for(int i = 0; i < SIZE; i++)
            System.out.print("=");

        System.out.print("+");
    }

    public static void printWindow() {
        for(int i = 0; i < SIZE; i++) {
            System.out.print("|");

            for(int j = 0; j < SIZE; j++)
                System.out.print(" ");

            System.out.print("|");

            for(int j = 0; j < SIZE; j++)
                System.out.print(" ");

            System.out.print("|");
            System.out.println();
        }
    }
}
```

Exercise: 2.20: StartFigure

Write a program in a class named StarFigure that produces the following output using for loops.

```
////////////////\\\\\\\\\\\\\\\\
////////////********\\\\\\\\\\\\
////////****************\\\\\\\\
////************************\\\\
********************************
```

```
public class StarFigure {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {

            for (int j = 1; j <= 20 - 4 * i; j++) {
                System.out.print("/");
            }

            for (int j = 1; j <= 8 * i - 8; j++) {
                System.out.print("*");
            }
        for (int j = 1; j <= 20 - 4 * i; j++) {
            System.out.print("\\");
        }

        System.out.println();
      }
   }
}
```

Exercise 2.21: StartFigure2

Modify your StarFigure code from the previous problem to use a constant for the size. The outputs below use a constant size of 3 (left) and 6 (right):

size 3	**size 6**
////////\\\\\\\\	////////////////////\\\\\\\\\\\\\\\\\\\\
////********\\\\	////////////////********\\\\\\\\\\\\\\\\
****************	////////////****************\\\\\\\\\\\\
	////////************************\\\\\\\\
	////********************************\\\\
	**

(You must solve this problem using only ONE public static final constant, not multiple constants; and its value must be used in the way described in this problem.)

```
public class StarFigure2 {
   public static final int R = 7;

                      public static void main(String[] args) {
                        for (int i = 1; i <= R; i++) {
        for (int j = 1; j <= (4*R) - 4 * i; j++) {
            System.out.print("/");

                        }
                        for (int j = 1; j <= 8 * i - 8; j++) {
        System.out.print("*");
                        }
                        for (int j = 1; j <= (4*R) - 4 * i; j++) {
        System.out.print("\\");
      }
                        System.out.println();
      }
```

```
    }
}
```

Exercise 2.22: DollarFigure

Write a Java program called DollarFigure that produces the following output. Use nested for loops to capture the structure of the figure.

```
$$$$$$$**************$$$$$$$
**$$$$$$************$$$$$$**
****$$$$$**********$$$$$****
******$$$$********$$$$******
********$$$******$$$********
**********$$****$$**********
************$**$************
```

```
public class DollarFigure {
    public static void main(String[] args) {
        for(int i = 1; i <= 7; i++) {
            for(int j = 0; j < 2 * i - 2; j++)
                System.out.print("*");

            for(int j = 0; j < -i + 8; j++)
                System.out.print("$");

            for(int j = 0; j < -2 * i + 16; j++)
                System.out.print("*");

            for(int j = 0; j < -i + 8; j++)
                System.out.print("$");

            for(int j = 0; j < 2 * i - 2; j++)
                System.out.print("*");

            System.out.println();
        }
    }
}
```

Exercise 2.23: DollarFigure2

Modify your DollarFigure program from the previous exercise to become a new program called DollarFigure2 that uses a global constant for the figure's height. (You may want to make loop tables first.) The previous output used a constant height of 7. The outputs below use a constant size of 3 (left) and 5 (right):

size 3	size 5
$$$******$$$	$$$$$**********$$$$$
$$****$$	**$$$$********$$$$**
****$**$****	****$$$******$$$****
	******$$****$$******
	********$**$********

(You must solve this problem using only ONE public static final constant, not multiple constants; and its value must be used in the way described in this problem.)

```
public class DollarFigure2 {
   public static final int R= 5;

         public static void main(String[] args) {
                  for(int i = 1; i <= R; i++) {
         for(int j = 0; j < 2 * i - 2; j++)
            System.out.print("*");
         for(int j = 0; j <= -i + R; j++)
            System.out.print("$");
         for(int j = 0; j <= -2 * i + (R*2+1); j++)
            System.out.print("*");
         for(int j = 0; j <= -i + R; j++)
            System.out.print("$");
         for(int j = 0; j < 2 * i - 2; j++)
            System.out.print("*");
         System.out.println();

         }
         }
}
```

Chapter 3: Parameters and Objects

Exercise 3.1: printNumber

Write a method called printNumbers that accepts a maximum number as a parameter and prints each number from 1 up to that maximum, inclusive, boxed by square brackets. For example, consider the following calls:

```
printNumbers(15);
printNumbers(5);
```

These calls should produce the following output:

```
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15]
[1] [2] [3] [4] [5]
```

You may assume that the value passed to printNumbers is 1 or greater.

```
public static void printNumbers(int max) {
   for (int i = 1; i <= max; i++) {
      System.out.print("[");
      System.out.print(i);
      System.out.print("] ");
   }

   System.out.println();
}
```

Exercise 3.2: printPowerOf2

Write a method called printPowersOf2 that accepts a maximum number as an argument and prints each power of 2 from 2^0 (1) up to that maximum power, inclusive. For example, consider the following calls:

```
printPowersOf2(3);
printPowersOf2(10);
```

These calls should produce the following output:

```
1 2 4 8
1 2 4 8 16 32 64 128 256 512 1024
```

You may assume that the value passed to printPowersOf2 is 0 or greater. (The Math class may help you with this problem. If you use it, you may need to cast its results from double to int so that you don't see a .0 after each number in your output. Also, can you write this program without using the Math class?)

```
public static void printPowersOf2(int max) {
    for (int i = 0; i <= max; i++) {
        System.out.print((int) Math.pow(2, i) + " ");
    }

    System.out.println();
}
```

Exercise 3.3: printPowerOfN

Write a method called printPowersOfN that accepts a base and an exponent as arguments and prints each power of the base from base^0 (1) up to that maximum power, inclusive. For example, consider the following calls:

```
printPowersOfN(4, 3);
printPowersOfN(5, 6);
printPowersOfN(-2, 8);
```

These calls should produce the following output:

```
1 4 16 64
1 5 25 125 625 3125 15625
1 -2 4 -8 16 -32 64 -128 256
```

You may assume that the exponent passed to printPowersOfN has a value of 0 or greater. (The Math class may help you with this problem. If you use it, you may need to cast its results from double to int so that you don't see a .0 after each number in your output. Also, can you write this program without using the Math class?)

```
public static void printPowersOfN(int base, int exponent) {
    int currentNum = 1;
    System.out.print(currentNum + " ");

    for (int i = 1; i <= exponent; i++) {
        currentNum *= base;
        System.out.print(currentNum + " ");
    }
    System.out.println();
}
```

Exercise 3.4: printSquare

Write a method called printSquare that takes in two integer parameters, a *min* and a *max*, and prints the numbers in the range from *min* to *max* inclusive in a square pattern. The square pattern is easier to understand by example than by explanation, so take a look at the sample method calls and their resulting console output in the table below.

Each line of the square consists of a circular sequence of increasing integers between *min* and *max*. Each line prints a different permutation of this sequence. The first line begins with *min*, the second line begins with *min* + 1, and so on. When the sequence in any line reaches *max*, it wraps around back to *min*.

You may assume the caller of the method will pass a min and a max parameter such that min is less than or equal to max.

Call	printSquare(1, 5);	printSquare(3, 9);	printSquare(0, 3);	printSquare(5, 5);
Output	12345 23451 34512 45123 51234	3456789 4567893 5678934 6789345 7893456 8934567 9345678	0123 1230 2301 3012	5

```
public static void printSquare(int min, int max) {
    for (int i = min; i <= max; i++) {

        for (int j = i; j <= max; j++) {
            System.out.print(j);
        }

        for (int j = min; j < i; j++) {
            System.out.print(j);
        }

        System.out.println();
    }
```

}

Exercise 3.5: printGrid
Write a method named printGrid that accepts two integer parameters *rows* and *cols*. The output is a comma-separated grid of numbers where the first parameter (*rows*) represents the number of rows of the grid and the second parameter (*cols*) represents the number of columns. The numbers count up from 1 to (*rows* x *cols*). The output are displayed in column-major order, meaning that the numbers shown increase sequentially down each column and wrap to the top of the next column to the right once the bottom of the current column is reached. Assume that *rows* and *cols* are greater than 0.

Here are some example calls to your method and their expected results:

Call:	printGrid(3, 6);	printGrid(5, 3);	printGrid(4, 1);	printGrid(1, 3);
Output:	1, 4, 7, 10, 13, 16 2, 5, 8, 11, 14, 17 3, 6, 9, 12, 15, 18	1, 6, 11 2, 7, 12 3, 8, 13 4, 9, 14 5, 10, 15	1 2 3 4	1, 2, 3

```
public static void printGrid(int row, int col) {

   for (int i = 1; i <= row; i++) {
      int num = i;
      System.out.print(i);

      for (int j = 1; j < col; j++) {
         num += row;
         System.out.print(", " + num);
      }

      System.out.println();
   }
}
```

Exercise 3.6: largerAbsVal
Write a method called largerAbsVal that takes two integers as parameters and returns the larger of the two absolute values. A call of largerAbsVal(11, 2) would return 11, and a call of largerAbsVal(4, -5) would return 5.

```
public static int largerAbsVal(int a, int b) {
   return Math.max(Math.abs(a), Math.abs(b));
}
```

Exercise 3.7: largestAbsVal
Write a method largestAbsVal that accepts three integers as parameters and returns the largest of their three absolute values. For example, a call of largestAbsVal(7, -2, -11) would return 11, and a call of largestAbsVal(-4, 5, 2) would return 5.

```
public static int largestAbsVal(int a, int b, int c) {
   return Math.max(Math.abs(a), Math.max(Math.abs(b), Math.abs(c)));
}
```

Exercise 3.8: quadratic
Write a method called quadratic that solves quadratic equations and prints their roots. Recall that a quadratic equation is a polynomial equation in terms of a variable *x* of the form $a x^2 + b x + c = 0$. The formula for solving a quadratic equation is:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Here are some example equations and their roots:

equation	$x^2 - 7x + 12$	$x^2 + 3x + 2$
call	quadratic(1, -7, 12);	quadratic(1, 3, 2);
output	First root = 4.0 Second root = 3.0	First root = -1.0 Second root = -2.0

Your method should accept the coefficients *a*, *b*, and *c* as parameters and should print the roots of the equation. You may assume that the equation has two real roots, though mathematically this is not always the case.

Also, there should be two roots, one the result of the addition, the other, the result of the subtraction. Print the root resulting from the addition first.

```
public static void quadratic(int a, int b, int c) {
   double squareRoot = Math.sqrt(b * b - 4 * a * c);
```

```
    double firstRoot = (-b + squareRoot) / (2 * a);
    double secondRoot = (-b - squareRoot) / (2 * a);
    System.out.println("First root = " + firstRoot);
    System.out.println("Second root = " + secondRoot);
}
```

Exercise 3.9: lastDigit

Write a method named lastDigit that returns the last digit of an integer. For example, lastDigit(3572) should return 2. It should work for negative numbers as well. For example, lastDigit(-947) should return 7.

Call	Value Returned
lastDigit(3572)	2
lastDigit(-947)	7
lastDigit(6)	6
lastDigit(35)	5
lastDigit(123456)	6

(Hint: This is a short method. You may not use a String to solve this problem.)

```
public static int lastDigit(int num) {
   return Math.abs(num % 10);
}
```

Exercise 3.10:

Write a method named area that accepts the radius of a circle as a parameter and returns the area of a circle with that radius. For example, the call area(2.0) should return 12.566370614359172. You may assume that the radius is non-negative.

```
public static double area(double radius) {
    return Math.PI * radius * radius;
}
```

Exercise 3.11: distance

Write a method called distance that accepts four integer coordinates *x1*, *y1*, *x2*, and *y2* as parameters and computes the distance between points (*x1*, *y1*) and (*x2*, *y2*) on the Cartesian plane. The equation for the distance is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

For example, the call of distance(1, 0, 4, 4) would return 5.0 and the call of distance(10, 2, 3, 5) would return 7.615773105863909.

```
public static double distance(int x1, int y1, int x2, int y2) {
   return Math.sqrt(Math.pow(x2 - x1, 2) + Math.pow(y2 - y1, 2));
}
```

Exercise 3.12: scientific
Write a method called scientific that accepts two real numbers as parameters for a base and an exponent and computes the base times 10 to the exponent, as seen in scientific notation. For example, the call of scientific(6.23, 5.0) would return 623000.0 and the call of scientific(1.9, -2.0) would return 0.019.

```
public static double scientific(double base, double exponent) {
   return base * Math.pow(10, exponent);
}
```

Exercise 3.13: pay
Write a method named pay that accepts a real number for a TA's salary and an integer for the number of hours the TA worked this week, and returns how much money to pay the TA. For example, the call pay(5.50, 6) should return 33.0.

The TA should receive "overtime" pay of 1 ½ normal salary for any hours above 8. For example, the call pay(4.00, 11) should return (4.00 * 8) + (6.00 * 3) or 50.0.

```
public static double pay(double hourSalary, int hour) {
   double salary = 0;
   if (hour > 8) {
      salary = 8 * hourSalary + (hour - 8) * (1.5 * hourSalary);
   } else {
      salary = hour * hourSalary;
   }
   return salary;
}
```

Exercise 3.14: cylinderSurfaceArea
Write a method called cylinderSurfaceArea that accepts a radius and height (both real numbers) as parameters and returns the surface area of a cylinder with those dimensions. For example, the call cylinderSurfaceArea(3.0, 4.5) should return 141.3716694115407. The formula for the surface area of a cylinder with radius *r* and height *h* is the following:

surface area = $2\pi r^2 + 2\pi rh$

```
public static double cylinderSurfaceArea(double radius, double height) {
   return 2 * Math.PI * radius * (radius + height);
}
```

Exercise 3.15: sphereVolume

Write a method called sphereVolume that accepts a radius (a real number) as a parameter and returns the volume of a sphere with that radius. For example, the call sphereVolume(2.0) should return 33.510321638291124. The formula for the volume of a sphere with radius *r* is the following:

volume = $^{4}/_{3}\ \pi\ r^3$

```
public static double sphereVolume(double radius) {
   return 4 * Math.PI * Math.pow(radius, 3) / 3;
}
```

Exercise 3.16: triangleArea

Write a method called triangleArea that accepts the three side lengths of a triangle (all real numbers) as parameters and returns the area of a triangle with those side lengths. For example, the call triangleArea(8, 5.2, 7.1) should return 18.151176098258745. To compute the area, use Heron's formula, which states that the area of a triangle whose three sides have lengths *a*, *b*, and *c*, is the following. The formula is based on the computed value *s*, a length equal to half the perimeter of the triangle:

area = $\sqrt{}$ (s (s-a)(s-b)(s-c))
where s = (a + b + c) / 2

```
public static double triangleArea(double a, double b, double c) {
   double s = (a + b + c) / 2;
   return Math.sqrt(s * (s - a) * (s - b) * (s - c));
}
```

Exercise 3.17:padString

Write a method padString that accepts two parameters: a String and an integer representing a length. The method should pad the parameter string with spaces until its length is the given length. For example, padString("hello", 8) should return "   hello". (This sort of method is useful when trying to print output that lines up horizontally.) If the string's length is already at least as long as the length parameter, your method should return the original string. For example, padString("congratulations", 10) would return "congratulations".

```
public static String padString(String string, int length) {
   int wordLength = string.length();
```

```
    for (int space = 1; space <= length - wordLength; space++) {
        string = " " + string;
    }

    return string;
}
```

Exercise 3.18: vertical

Write a method called vertical that accepts a String as its parameter and prints each letter of the string on separate lines. For example, a call of vertical("hey now") should produce the following output:

```
h
e
y

n
o
w
```

```
public static void vertical(String string) {

    for (int i = 0; i < string.length(); i++) {
        System.out.println(string.charAt(i));
    }
}
```

Exercise 3.19: printReverse

Write a method called printReverse that accepts a String as its parameter and prints the characters in opposite order. For example, a call of printReverse("hello there!"); should print the following output:

```
!ereht olleh
```

If the empty string is passed, no output is produced. Your method should produce a complete line of output.

```
public static void printReverse(String string) {

    for (int i = string.length() - 1; i >= 0; i--) {
        System.out.print(string.charAt(i));
    }
}
```

Exercise 3.20: inputBirthday

Write a method called inputBirthday that accepts a Scanner for the console as a parameter and prompts the user to enter a month, day, and year of birth, then prints the birthdate in a suitable format. Here is an example dialogue with the user:

```
On what day of the month were you born? 8
What is the name of the month in which you were born? May
During what year were you born? 1981
You were born on May 8, 1981. You're mighty old!
```

```
public static void inputBirthday(Scanner console) {
    System.out.print("On what day of the month were you born? ");
    int day = console.nextInt();

    System.out.print("What is the name of the month in which you were born? ");
    String month = console.next();

    System.out.print("During what year were you born? ");
    int year = console.nextInt();

    System.out.println("You were born on " + month + " " + day + ", " + year + ". You're mighty o
ld!");
}
```

Exercise 3.21: processName

Write a method called processName that accepts a Scanner for the console as a parameter and that prompts the user to enter his or her full name, then prints the name in reverse order (i.e., last name, first name). You may assume that only a first and last name will be given. You should read the entire line of input at once with the Scanner and then break it apart as necessary. Here is a sample dialogue with the user:

```
Please enter your full name: Sammy Jankis
Your name in reverse order is Jankis, Sammy
```

```
public static void processName(Scanner console) {
   System.out.print("Please enter your full name: ");
   String firstName = console.next();
```

```
  String lastName = console.next();
  System.out.print("Your name in reverse order is " + lastName + ", " + firstName);
}
```

Exercise 3.22: TheNameGameExercise

Write a complete program called "TheNameGame", where the user inputs a first and last name and a song in the following format is printed about their first, then last, name. Use a method to avoid redundancy.

```
What is your name? **Fifty Cent**
Fifty Fifty, bo-Bifty
Banana-fana fo-Fifty
Fee-fi-mo-Mifty
FIFTY!

Cent Cent, bo-Bent
Banana-fana fo-Fent
Fee-fi-mo-Ment
CENT!
```

```
public class TheNameGame {
        public static void main(String[] args){
          Scanner scan = new Scanner(System.in);
          System.out.print("What is your name? ");
          String first = scan.next();
          String last = scan.next();

         method(first);
         System.out.println();
          method(last);

   }

   public static void method(String name){
      System.out.println(name + " " + name + ", bo-" + name.replace(name.charAt(0), 'B'));
      System.out.println("Banana-fana fo-" + name.replace(name.charAt(0), 'F'));
      System.out.println("Fee-fi-mo-" + name.replace(name.charAt(0), 'M'));
      System.out.println(name.toUpperCase() + "!");
   }
}
```

Chapter 4: Conditional Execution

Exercise 4.1: fractionSum
Write a method called fractionSum that accepts an integer parameter *n* and returns as a double the sum of the first *n* terms of the sequence:

$$\sum_{i=1}^{n} \frac{1}{i}$$

In other words, the method should generate the following sequence:

1 + (1/2) + (1/3) + (1/4) + (1/5) + ...

You may assume that the parameter *n* is non-negative.

```
public static double fractionSum(int n) {
   double sum = 0.0;

   for (double i = 1.0; i <= n; i++) {
      sum += 1 / i;
   }

   return sum;
}
```

Exercise 4.2: repl
Write a method named repl that accepts a String and a number of repetitions as parameters and returns the String concatenated that many times. For example, the call repl("hello", 3) returns "hellohellohello". If the number of repetitions is 0 or less, an empty string is returned.

```
public static String repl(String word, int rept) {
   String concatenation = "";

   for (int i = 0; i < rept; i++) {
      concatenation += word;
   }

   return concatenation;
}
```

Exercise 4.3: season

Write a method named season that takes two integers as parameters representing a month and day and that returns a String indicating the season for that month and day. Assume that months are specified as an integer between 1 and 12 (1 for January, 2 for February, and so on) and that the day of the month is a number between 1 and 31.

If the date falls between 12/16 and 3/15, you should return "Winter". If the date falls between 3/16 and 6/15, you should return "Spring". If the date falls between 6/16 and 9/15, you should return "Summer". And if the date falls between 9/16 and 12/15, you should return "Fall".

```
public static String season(int month, int day) {
   int days = (month - 1) * 31 + day;
   if (days >= 78 && days <= 170) {
      return "Spring";
   } else if (days >= 171 && days <= 263) {
      return "Summer";
   } else if (days >= 264 && days <= 356) {
      return "Fall";
   }else {
      return "Winter";
   }
}
```

Exercise 4.4: daysInMonth

Write a method named daysInMonth that accepts a month (an integer between 1 and 12) as a parameter and returns the number of days in that month in this year. For example, the call daysInMonth(9) would return 30 because September has 30 days. Assume that the code is not being run during a leap year (that February always has 28 days).

Month	1 Jan	2 Feb	3 Mar	4 Apr	5 May	6 Jun	7 Jul	8 Aug	9 Sep	10 Oct	11 Nov	12 Dec
Days	31	28	31	30	31	30	31	31	30	31	30	31

```
public static int daysInMonth(int month) {
   if (month == 2) {
      return 28;
   } else if (month == 4 || month == 6 || month == 9 || month == 11) {
      return 30;
   } else {
      return 31;
   }
}
```

Exercise 4.5: pow

Write a method named pow that accepts a base and an exponent as parameters and returns the base raised to the given power. For example, the call pow(3, 4) returns 3 * 3 * 3 * 3 or 81. Do not use Math.pow in your solution. Assume that the base and exponent are non-negative.

```
public static int pow(int base, int exp) {
   int product = 1;

   for (int i = 0; i < exp; i++) {
      product *= base;
   }

   return product;
}
```

Exercise 4.6: printRange

Write a method called printRange that accepts two integers as arguments and prints the sequence of numbers between the two arguments, separated by spaces. Print an increasing sequence if the first argument is smaller than the second; otherwise, print a decreasing sequence. If the two numbers are the same, that number should be printed by itself. Here are some sample calls to printRange:

```
printRange(2, 7);
printRange(19, 11);
printRange(5, 5);
```

The output produced should be the following:

```
2 3 4 5 6 7
19 18 17 16 15 14 13 12 11
5
```

```
public static void printRange(int a, int b) {
   if (a <= b) {
      for (int i = a; i <= b; i++) {
         System.out.print(i + " ");
      }
   } else {
      for (int i = a; i >= b; i--) {
         System.out.print(i + " ");
      }
   }
   System.out.println();
}
```

Exercise 4.7: xo

Write a static method named xo that accepts an integer *size* as a parameter and prints a square of *size* by *size* characters, where all characters are "o" except that an "x" pattern of "x" characters has been drawn from the corners of the square. In other words, on the first line, the first and last characters are "x"; on the second line, the second and second-from-last characters are "x"; and so on. If 0 or less is passed for the size, no output should be produced.

The following table lists some calls to your method and their expected output:

Call	xo(5);	xo(8);	xo(3);	xo(1);	xo(0);	xo(12);	xo(11);
Output	xooox	xoooooox	xox	x		xoooooooooox	xooooooooox
	oxoxo	oxooooxo	oxo			oxooooooooxo	oxoooooooxo
	ooxoo	ooxooxoo	xox			ooxooooooxoo	ooxoooooxoo
	oxoxo	oooxxooo				oooxooooxooo	oooxoooxooo
	xooox	oooxxooo				ooooxooxoooo	ooooxoxoooo
		ooxooxoo				oooooxxooooo	oooooxooooo
		oxooooxo				oooooxxooooo	ooooxoxoooo
		xoooooox				ooooxooxoooo	oooxoooxooo
						oooxooooxooo	ooxoooooxoo
						ooxooooooxoo	oxoooooooxo
						oxooooooooxo	xooooooooox
						xoooooooooox	

```
public static void xo(int size) {

    for (int line = 1; line <= size; line++) {
        if (line > (size + 1) / 2) {
            for (int o = 0; o < size - line; o++) {
                System.out.print("o");
            }
        } else {
            for (int o = 0; o < line - 1; o++) {
               System.out.print("o");
            }
        }
        System.out.print("x");

        if (line > (size + 1) / 2) {
            for (int o = 0; o < 2 * line - size - 2; o++) {
                System.out.print("o");
            }
        } else {
            for (int o = 0; o < size - 2 * line; o++) {
                System.out.print("o");
            }
        }

        if (size % 2 == 0 || line != size / 2 + 1) {
            System.out.print("x");
        }

        if (line > (size + 1) / 2) {
            for (int o = 0; o < size - line; o++) {
                System.out.print("o");
            }
        } else {
            for (int o = 0; o < line - 1; o++) {
                System.out.print("o");
            }
        }

        System.out.println();
    }
}
```

Exercise 4.8: smallestLargest

Write a method named smallestLargest that asks the user to enter numbers, then prints the smallest and largest of all the numbers typed in by the user. You may assume the user enters a valid number greater than 0 for the number of numbers to read. Here is an example dialogue:

```
How many numbers do you want to enter? 4
Number 1: 5
Number 2: 11
Number 3: -2
Number 4: 3
Smallest = -2
Largest = 11
```

```
public static void smallestLargest() {
    Scanner console = new Scanner(System.in);
    System.out.print("How many numbers do you want to enter? ");
    int number = console.nextInt();
    System.out.print("Number 1: ");
    int firstNum = console.nextInt();
    int min = firstNum;
    int max = firstNum;

    for (int i = 2; i <= number; i++) {
        System.out.print("Number " + i + ": ");
        int num = console.nextInt();
        if (num < min) {
            min = num;
        } else if (num > max) {
            max = num;
        }
    }

    System.out.printf("Smallest = %d\n", min);
    System.out.printf("Largest = %d\n", max);
}
```

Exercise 4.9: evenSumMAx

Write a method named evenSum that prompts the user for many integers and print the total even sum and maximum of the even numbers. You may assume that the user types at least one non-negative even integer.

```
how many integers? 4
next integer? 2
next integer? 9
next integer? 18
next integer? 4
even sum = 24
even max = 18
```

```
public static void evenSum() {
    Scanner console = new Scanner(System.in);
    System.out.print("how many integers? ");
    int numInt = console.nextInt();
    int max = 0;
    int sum = 0;

    for (int i = 0; i < numInt; i++) {
        System.out.print("next integer? ");
        int integer = console.nextInt();
        if (integer % 2 == 0) {
            sum += integer;
            if (integer > max) {
                max = integer;
            }
        }
    }

    System.out.println("even sum = " + sum);
    System.out.println("even max = " + max);
}
```

Exercise 4.10: printGPA

Write a method named printGPA that calculates a student's grade point average. The user will type a line of input containing the student's name, then a number of scores, followed by that many integer scores. Here are two example dialogues:

```
Enter a student record: Maria 5 72 91 84 89 78
Maria's grade is 82.8
Enter a student record: Jordan 4 86 71 62 90
Jordan's grade is 77.25
```

For example, Maria's grade is 82.8 because her average of (72 + 91 + 84 + 89 + 78) / 5 equals 82.8. Use a Scanner for user input.

```
public static void printGPA() {
    Scanner console = new Scanner(System.in);
    System.out.print("Enter a student record: ");
    String name = console.next();
    int numScore = console.nextInt();
    double sum = 0.0;

    for (int i = 0; i < numScore; i++) {
        int score = console.nextInt();
        sum += score;
    }

    double average = sum / numScore;
    System.out.println(name + "'s grade is " + average);
}
```

Exercise 4.11: lognestName

Write a static method named longestName that reads names typed by the user and prints the longest name (the name that contains the most characters) in the format shown below. Your method should accept a console Scanner and an integer *n* as parameters and should then prompt for *n* names.

The longest name should be printed with its first letter capitalized and all subsequent letters in lowercase, regardless of the capitalization the user used when typing in the name. If there is a tie for longest between two or more names, use the tied name that was typed earliest. Also print a message saying that there was a tie, as in the right log below. It's possible that some shorter names will tie in length, such as DANE and Erik in the left log below; but don't print a message unless the tie is between the longest names. You may assume that *n* is at least 1, that each name is at least 1 character long, and that the user will type single-word names consisting of only letters. The following table shows two sample calls and their output (user input in blue):

Call	`Scanner console = new Scanner(System.ir`
	`longestName(console, 5);`
		`Scanner console = new Scanner(System.ir`
		`longestName(console, 7);`

Output	name #1? **roy** name #2? **DANE** name #3? **Erik** name #4? **sTeFaNiE** name #5? **LaurA** Stefanie's name is longest	name #1? **TrEnt** name #2? **rita** name #3? **JORDAN** name #4? **craig** name #5? **leslie** name #6? **YUKI** name #7? **TaNnEr** Jordan's name is longest (There was a tie!)

```
public static void longestName(Scanner console, int numOfNames) {
    System.out.print("name #1? ");
    String name = console.next();
    boolean tie = false;
    int maxLength = name.length();
    String longestName = name.toLowerCase();

    for (int i = 2; i <= numOfNames; i++) {
        System.out.printf("name #%d? ", i);
        name = console.next();
        if (name.length() > maxLength) {
            maxLength = name.length();
            longestName = name.toLowerCase();
            tie = false;
        } else if (name.length() == maxLength){
            tie = true;
        }
    }
    longestName = longestName.substring(0,1).toUpperCase() + longestName.substring(1);
    System.out.println(longestName + "'s name is longest");

    if (tie == true) {
        System.out.println("(There was a tie!)");
    }
}
```

Exercise 4.12: printTriangleType

Write a method called printTriangleType that accepts three integer arguments representing the lengths of the sides of a triangle and prints what type of triangle it is. The three types are equilateral, isosceles, and scalene. An equilateral triangle has all three sides the same length, an isosceles triangle has two sides the same length, and a scalene triangle has three sides of different lengths. Here are some example calls to printTriangleType:

```
printTriangleType(5, 7, 7);
printTriangleType(6, 6, 6);
printTriangleType(5, 7, 8);
printTriangleType(12, 18, 12);
```

The output produced should be the following:

```
isosceles
equilateral
scalene
isosceles
```

```
public static void printTriangleType(int a, int b, int c) {
    if (a == b && b == c) {
        System.out.println("equilateral");
    } else if (a == b || b == c || a == c) {
        System.out.println("isosceles");
    } else {
        System.out.println("scalene");
    }
}
```

Exercise 4.13: average

Write a method called average that takes two integers as parameters and returns the average of the two integers.

```
public static double average(int a, int b) {
   return (a + b) / 2.0;
}
```

Exercise 4.14: pow2
Write a method named pow2 (a variation of the previous pow exercise) that accepts a real number base and an integer exponent as parameters and returns the base raised to the given power. Your code should work for both positive and negative exponents. For example, the call pow2(2.0, -2) returns 0.25. Do not use Math.pow in your solution.

```
public static double pow2(double base, int exponent) {
   double result = 1.0;

   if (exponent >= 0) {
      for (int i = 0; i < exponent; i++) {
         result *= base;
      }
   } else {
      for (int i = exponent; i < 0; i++) {
         result *= 1 / base;
      }
   }

   return result;
}
```

Exercise 4.15: getGrade
Write a method called getGrade that accepts an integer representing a student's grade in a course and returns that student's numerical course grade. The grade can be between 0.0 (failing) and 4.0 (perfect). Assume that scores are in the range of 0 to 100 and that grades are based on the following scale:

Score	**Grade**
<60	0.0
60-62	0.7
63	0.8
64	0.9
65	1.0
...	
92	3.7
93	3.8

94	3.9
>=95	4.0

For an added challenge, make your method throw an IllegalArgumentException if the user passes a grade lower than 0 or higher than 100.

```
public static double getGrade(int score) {
   if (score < 0 || score > 100) {
      throw new IllegalArgumentException();
   } else {
      if (score < 60) {
         return 0.0;
      } else if (score <= 62) {
         return 0.7;
      } else if (score <= 94) {
         double grade = 0.8;

         for(int i = 63; i < score; i++){
            grade += 0.1;
         }

         return grade;
      } else {
         return 4.0;
      }
   }
}
```

Exercise 4.16: printPalindrome

Write a method called printPalindrome that accepts a Scanner for the console as a parameter, and prompts the user to enter one or more words and prints whether the entered String is a palindrome (i.e., reads the same forwards as it does backwards, like "abba" or "racecar"). If the following Scanner object were declared:

```
Scanner console = new Scanner(System.in);
printPalindrome(console);
```

The resulting output for a call where the user types a palindrome would be:

```
Type one or more words: racecar
racecar is a palindrome!
```

The output for a call where the user types a word that is not a palindrome would be:

```
Type one or more words: hello
hello is not a palindrome.
```

For an added challenge, make the code case-insensitive, so that words like "Abba" and "Madam" will be considered palindromes.

```
public static void printPalindrome(Scanner console) {
                System.out.print("Type one or more words: ");
                String word = console.nextLine();
                String string = word.toLowerCase();
                int i = 0;
                int j = word.length() - 1;
                while(i < j) {
                if(string.charAt(i) != string.charAt(j)) {
                        System.out.println(word + " is not a palindrome.");
                        return;
                }
                        i++;
                        j--;
                }
                        System.out.println(word + " is a palindrome!");
        }
```

Exercise 4.17: stutter
Write a method called stutter that accepts a parameter and returns the String with its characters returns repeated twice. For example, stutter("Hello!") returns "HHeellllloo!!"

```
public static String stutter(String string) {
                String s = "";
                for(int i = 0; i < string.length(); i++) {
                        s = s + string.charAt(i) + string.charAt(i);

                }
                return s;
        }
```

Exercise 4.18: wordCount
Write a method called wordCount that accepts a String as its parameter and returns the number of words in the String. A word is a sequence of one or more nonspace characters (any character other than ' '). For example, the call wordCount("hello") should return 1, the call wordCount("how are you?") should return 3, the call wordCount(" this   string has   wide   spaces ") should return 5, and the call wordCount(" ") should return 0.

```
public static int wordCount(String string) {
   int numOfWords = 0;
   char firstChar = string.charAt(0);

   if (firstChar != ' ') {
      numOfWords++;
   }

   for (int i = 1; i < string.length(); i++) {
      char secondChar = string.charAt(i);
      if (firstChar == ' ' && secondChar != ' ') {
         numOfWords++;
      }
      firstChar = secondChar;
   }

   return numOfWords;
}
```

Exercise 4.19: quadrant

Write a static method called quadrant that takes as parameters a pair of real numbers representing an (x, y) point and that returns the quadrant number for that point. Recall that quadrants are numbered as integers from 1 to 4 with the upper-right quadrant numbered 1 and the subsequent quadrants numbered in a counter-clockwise fashion:

```
                  ^ y-axis
                  |
                  |
                  |
      Quadrant 2  |  Quadrant 1
                  |
<-----------------+-----------------> x-axis
                  |
      Quadrant 3  |  Quadrant 4
                  |
                  |
                  |
                  V
```

Notice that the quadrant is determined by whether the x and y coordinates are positive or negative numbers. If a point falls on the x-axis or the y-axis, then the method should return 0. Below are sample calls on the method.

Call	Value Returned
quadrant(12.4, 17.8)	1
quadrant(-2.3, 3.5)	2
quadrant(-15.2, -3.1)	3
quadrant(4.5, -42.0)	4
quadrant(0.0, 3.14)	0

```
public static int quadrant(double x, double y) {
   if (x > 0 && y > 0) {
      return 1;
   } else if (x < 0 && y > 0) {
      return 2;
   } else if (x < 0 && y < 0) {
      return 3;
   } else if (x > 0 && y < 0) {
      return 4;
   } else {
      return 0;
   }
}
```

Exercise 4.20: numUnique
Write a method named numUnique that takes three integers as parameters and that returns the number of unique integers among the three. For example, the call numUnique(18, 3, 4) should return 3 because the parameters have 3 different values. By contrast, the call numUnique(6, 7, 6) would return 2 because there are only 2 unique numbers among the three parameters: 6 and 7.

```
public static int numUnique(int a, int b, int c) {
   int count = 0;

   if (a != b) {
      count++;
   }

   if (b != c) {
      count++;
   }

   if (a != c) {
      count++;
   }

   if (a == b && b == c) {
      count++;
   }

   return count;
}
```

Exercise 4.21: perfectNumbers
A "perfect number" is a positive integer that is the sum of all its proper factors (that is, factors including 1 but not the number itself). The first two perfect numbers are 6 and 28, since 1+2+3=6 and 1+2+4+7+14=28. Write a static method perfectNumbers that takes an integer max as an argument and prints out all perfect numbers that are less than or equal to max.

Here is the console output from a call to perfectNumbers(6):

```
Perfect numbers up to 6: 6
```

Here is the console output from a call to perfectNumbers(500):

```
Perfect numbers up to 500: 6 28 496
```

```
public static void perfectNumbers(int max) {
   String numbers = "";

   for (int i = 1; i <= max; i++) {
      int sum = 0;

      for (int j = 1; j < i; j++) {
         if (i % j == 0) {
            sum += j;
         }
      }

      if (sum == i) {
         numbers += i + " ";
      }
   }

   System.out.println("Perfect numbers up to " + max + ": " + numbers);
}
```

Chapter 5: Program Logic and Indefinite Loops

Exercise 5.1: showTwos

Write a method named showTwos that shows the factors of 2 in a given integer. For example, the following calls:

```
showTwos(7);
showTwos(18);
showTwos(68);
showTwos(120);
```

should produce this output:

```
7 = 7
18 = 2 * 9
68 = 2 * 2 * 17
120 = 2 * 2 * 2 * 15
```

```
public static void showTwos(int num) {
   System.out.print(num + " = ");

   while (num % 2 == 0) {
        System.out.print("2 * ");
        num /= 2;
   }

   System.out.print(num);
}
```

Exercise 5.2: gcd

Write a method named gcd that accepts two integers as parameters and returns the greatest common divisor of the two numbers. The greatest common divisor (GCD) of two integers a and b is the largest integer that is a factor of both a and b. The GCD of any number and 1 is 1, and the GCD of any number and 0 is that number.

One efficient way to compute the GCD of two numbers is to use Euclid's algorithm, which states the following:

GCD(A, B) = GCD(B, A % B)
GCD(A, 0) = Absolute value of A

In other words, if you repeatedly mod A by B and then swap the two values, eventually B will store 0 and A will store the greatest common divisor.

For example: gcd(24, 84) returns 12, gcd(105, 45) returns 15, and gcd(0, 8) returns 8.

```
public static int gcd(int a, int b) {

   while (b != 0) {
      int c = a % b;
      a = b;
      b = c;
   }

   return Math.abs(a);
}
```

Exercise 5.3: toBinary
Write a method named toBinary that accepts an integer as a parameter and returns a string of that number's representation in binary. For example, the call of toBinary(42) should return "101010".

```
public static String toBinary(int num) {
   String result = "";
   result = num % 2 + result;
   num /= 2;

   while (num != 0) {
      result = num % 2 + result;
      num /= 2;
   }

   return result;
}
```

Exercise 5.4: randomX

Write a method named randomX that keeps printing lines, where each line contains a random number of x characters between 5 and 19 inclusive, until it prints a line with 16 or more characters. For example, the output from your method might be the following. Notice that the last line has 17 x characters.

```
xxxxxxx
xxxxxxxxxxxxx
xxxxxx
xxxxxxxxxxx
```

```
xxxxxxxxxxxxxxxxx
```

```
public static void randomX() {
   Random random = new Random();
   int x = random.nextInt(15) + 5;

   while (x < 15) {

      for (int i = 0; i < x; i++) {
         System.out.print("x");
      }

      System.out.println();
      x = random.nextInt(15) + 5;
   }

   for (int i = 0; i < x; i++) {
      System.out.print("x");
   }

   System.out.println();
}
```

Exercise 5.5: randomLines

Write a method called randomLines that prints between 5 and 10 random strings of letters (between "a" and "z"), one per line. Each string should have random length of up to 80 characters.

```
rlcuhubm
ilons
ahidbxonunonheuxudxrcgdp
xmkmkmmmmwmwqjbaeeeerceheelciheihcreidercdeehiuhhhn
hdcrphdidcrydxgtkdhoendgcidgxfidgfufdgfuuuuuu
```

(Because this problem uses random numbers, our test cases check only the general format of your output. You must still examine the output yourself to make sure the answer is correct.)

```
public static void randomLines() {
   Random random = new Random();
   int line = random.nextInt(5) + 5;

   for (int i = 0; i < line; i++) {
      int numOfChar = 0;
```

```
        while (numOfChar <= 0) {
            int letter = random.nextInt(26) + 97;
            System.out.print((char)letter);
            numOfChar++;
        }

        System.out.println();
    }
}
```

Exercise 5.6: makeGuesses

Write a method named makeGuesses that will guess numbers between 1 and 50 inclusive until it makes a guess of at least 48. It should report each guess and at the end should report the total number of guesses made. Below is a sample execution:

```
guess = 43
guess = 47
guess = 45
guess = 27
guess = 49
total guesses = 5
```

(Because this problem uses random numbers, our test cases check only the general format of your output. You must still examine the output yourself to make sure the answer is correct.)

```
public static void makeGuesses() {
    int numOfGuesses = 1;
    Random random = new Random();
    int guess = random.nextInt(50) + 1;
    System.out.println("guess = " + guess);

    while (guess < 48) {
        guess = random.nextInt(50) + 1;
        numOfGuesses++;
        System.out.println("guess = " + guess);
    }

    System.out.println("total guesses = " + numOfGuesses);
}
```

Exercise 5.7: diceSum

Write a method named diceSum that prompts the user for a desired sum, then repeatedly rolls two six-sided dice until their sum is the desired sum. Here is the expected dialogue with the user:

```
Desired dice sum: 9
4 and 3 = 7
3 and 5 = 8
5 and 6 = 11
5 and 6 = 11
1 and 5 = 6
6 and 3 = 9
```

(Because this problem uses random numbers, our test cases check only the general format of your output. You must still examine the output yourself to make sure the answer is correct.)

```
public static void diceSum() {
    Scanner console = new Scanner(System.in);
    System.out.print("Desired dice sum: ");
    int desiredSum = console.nextInt();
    int die1 = 0;
    int die2 = 0;
    Random r = new Random();
    while (die1 + die2 != desiredSum) {
        die1 = r.nextInt(6) + 1;
        die2 = r.nextInt(6) + 1;
        int sum = die1 + die2;
        System.out.println(die1 + " and " + die2 + " = " + (die1 + die2));
    }
}
```

Exercise 5.8: randomWalk

Write a method named randomWalk that performs a random one-dimensional walk, reporting each position reached and the maximum position reached during the walk. The random walk should begin at position 0. On each step, you should either increase or decrease the position by 1 (with equal probability). The walk stops when 3 or -3 is hit. The output should look like this:

```
position = 0
position = 1
position = 0
position = -1
position = -2
position = -1
position = -2
position = -3
max position = 1
```

```
public static void randomWalk() {
    int n = 0;
    int max = 0;
    Random r = new Random();
    System.out.println("position = " + n);
    while (-3 < n && n < 3) {
        int flip = r.nextInt(2);
        if (flip == 0) {
            n++;
        } else {
            n--;
        }
        max = Math.max(n, max);
        System.out.println("position = " + n);
    }
    System.out.println("max position = " + max);
}
```

Exercise 5.9: printFactors

Write a method named printFactors that accepts an integer as its parameter and uses a fencepost loop to print the factors of that number, separated by the word " and ". For example, the number 24's factors should print as:

```
1 and 2 and 3 and 4 and 6 and 8 and 12 and 24
```

You may assume that the number parameter's value is greater than 0.

```
// Prints the factors of an integer separated by " and ".
// Precondition: n >= 1
public static void printFactors(int n) {
    // print first factor, always 1 (fencepost)
    System.out.print(1);
    // print remaining factors, if any, preceded by " and "
    for (int i = 2; i <= n; i++) {
        if (n % i == 0) {
            System.out.print(" and " + i);
        }
    }
    // end the line of output
    System.out.println();
}
```

Exercise 5.10: hopscotch

Write a method named hopscotch that accepts an integer parameter for a number of "hops" and prints a hopscotch board of that many hops. A "hop" is defined as the split into two numbers and then back together again into one. For example, hopscotch(3); should print:

```
  1
2   3
  4
5   6
  7
8   9
  10
```

```
public static void hopscotch(int hops) {
    System.out.println("   1");
    for (int i = 1; i <= hops; i++) {
        System.out.println((3 * i - 1) + "     " + 3 * i);
        System.out.println("   " + (3 * i + 1));
    }
}
```

Exercise 5.11: threeHEads

Write a method named threeHeads that repeatedly flips a coin until three heads *in a row* are seen. You should use a Random object to give an equal chance to a head or a tail appearing. Each time the coin is flipped, what is seen is displayed (H for heads, T for tails). When 3 heads in a row are flipped a congratulatory message is printed. Here are possible outputs of two calls to threeHeads:

```
T T T H T H H H
Three heads in a row!
```

(Because this problem uses random numbers, our test cases check only the general format of your output. You must still examine the output yourself to make sure the answer is correct.)

```
public static void threeHeads() {
    Random rand = new Random();
    int heads = 0;
    while (heads < 3) {
        int flip = rand.nextInt(2);  // flip coin
        if (flip == 0) {            // heads
            heads++;
            System.out.print("H ");
        } else {                    // tails
            heads = 0;
            System.out.print("T ");
        }
    }
    System.out.println();
    System.out.println("Three heads in a row!");
}
```

Exercise 5.12: printAverage

Write a method named printAverage that accepts a Scanner for the console as a parameter and repeatedly prompts the user for numbers. Once any number less than zero is typed, the average of all non-negative numbers typed is displayed. Display the average as a double, and do not round it. For example, a call to your method might look like this:

```
Scanner console = new Scanner(System.in);
printAverage(console);
```

The following is one example log of execution for your method:

```
Type a number: 7
Type a number: 4
Type a number: 16
Type a number: -4
Average was 9.0
```

If the first number typed is negative, do not print an average. For example:

```
Type a number: -2
```

```
public static double printAverage(Scanner console) {
    System.out.print("Type a number: ");

    int input = console.nextInt();
    double sum = 0.0;
    int count = 0;
    while (input >= 0) {
        count++;
        sum = sum + input;
        System.out.print("Type a number: ");
        input = console.nextInt();
    }
    if (count > 0) {
        double average = sum / count;
        System.out.println("Average was " + average);
    }
    return sum;
}
```

Exercise 5.13: consecutive

Write a method named consecutive that accepts three integers as parameters and returns true if they are three consecutive numbers; that is, if the numbers can be arranged into an order such that there is some integer k such that the parameters' values are k, k+1, and k+2. Your method should return false if the integers are not consecutive. Note that order is not significant; your method should return the same result for the same three integers passed in any order.

For example, the calls consecutive(1, 2, 3), consecutive(3, 2, 4), and consecutive(-10, -8, -9) would return true. The calls consecutive(3, 5, 7), consecutive(1, 2, 2), and consecutive(7, 7, 9) would return false.

```
public static boolean consecutive(int a, int b, int c) {
    int min = Math.min(a, Math.min(b, c));
    int max = Math.max(a, Math.max(b, c));
    int mid = a + b + c - max - min;
    return min + 1 == mid && mid + 1 == max;
}
```

Exercise 5.14: hasMidpoint

Write a method named hasMidpoint that accepts three integers as parameters and returns true if one of the integers is the midpoint between the other two integers; that is, if one integer is exactly halfway between them. Your method should return false if no such midpoint relationship exists.

The integers could be passed in any order; the midpoint could be the 1st, 2nd, or 3rd. You must check all cases.

Calls such as the following should return true:

```
hasMidpoint(4, 6, 8)
hasMidpoint(2, 10, 6)
hasMidpoint(8, 8, 8)
hasMidpoint(25, 10, -5)
```

Calls such as the following should return false:

```
hasMidpoint(3, 1, 3)
hasMidpoint(1, 3, 1)
hasMidpoint(21, 9, 58)
hasMidpoint(2, 8, 16)
```

```
public static boolean hasMidpoint(int a, int b, int c) {
   double mid = (a + b + c) / 3.0;
   return (a == mid || b == mid || c == mid);
}
```

Exercise 5.15: dominant

Write a method dominant that accepts three integers as parameters and returns true if any one of the three integers is larger than the sum of the other two integers. The integers might be passed in any order, so the largest value could be any of the three. If no value is larger than the sum of the other two, your method should return false.

For example, the call of dominant(4, 9, 2) would return true because 9 is larger than 4 + 2. The call of dominant(5, 3, 7) would return false because none of those three numbers is larger than the sum of the others. You may assume that none of the numbers is negative.

```
public static boolean dominant(int a, int b, int c) {
   return a > b + c || b > a + c || c > a + b;
}
```

Exercise 5.16: anglePairs

Write a static method named anglePairs that accepts three angles (integers), measured in degrees, as parameters and returns whether or not there exists both complementary and supplementary angles amongst the three angles passed. Two angles are *complementary* if their sum is exactly 90 degrees; two angles are *supplementary* if their sum is exactly 180 degrees. Therefore, the method should return true if any two of the three angles add up to 90 degrees and also any two of the three angles add up to 180 degrees; otherwise the method should return false. You may assume that each angle passed is non-negative.

Here are some example calls to the method and their resulting return values.

Call	**Value Returned**
anglePairs(0, 90, 180)	true

anglePairs(45, 135, 45)	true
anglePairs(177, 87, 3)	true
anglePairs(120, 60, 30)	true
anglePairs(35, 60, 30)	false
anglePairs(120, 60, 45)	false
anglePairs(45, 90, 45)	false
anglePairs(180, 45, 45)	false

```
// boolean zen solution
public static boolean anglePairs(int a1, int a2, int a3) {
   return (a1 + a2 == 90 || a2 + a3 == 90 || a3 + a1 == 90) &&
          (a1 + a2 == 180 || a2 + a3 == 180 || a3 + a1 == 180);
}
```

Exercise 5.17: monthApart

Write a method named monthApart that accepts four integer parameters representing two calendar dates. Each date consists of a month (1 through 12) and a day (1 through the number of days in that month [28-31]). The method returns whether the dates are at least a month apart. Assume that all dates in this problem occur during the same year. For example, the following dates are all considered to be at least a month apart from 9/19 (September 19): 2/14, 7/25, 8/2, 8/19, 10/19, 10/20, and 11/5. The following dates are NOT at least a month apart from 9/19: 9/20, 9/28, 10/1, 10/15, and 10/18. Note that the first date could come before or after (or be the same as) the second date. Assume that all parameter values passed are valid.

Sample calls:

Call	Returns	Because
monthApart(6, 14, 9, 21)	true	June 14th is at least a month before September 21st
monthApart(4, 5, 5, 15)	true	April 5th is at least a month before May 15th
monthApart(4, 15, 5, 15)	true	April 15th is at least a month before May 15th
monthApart(4, 16, 5, 15)	false	April 16th is NOT at least a month before May 15th
monthApart(6, 14, 6, 8)	false	June 14th is NOT at least a month apart from June 8th
monthApart(7, 7, 6, 8)	false	July 7th is NOT at least a month apart from June 8th
monthApart(7, 8, 6, 8)	true	July 8th is at least a month after June 8th
monthApart(10, 14, 7, 15)	true	October 14th is at least a month after July 15th

```
public static boolean monthApart(int m1, int d1, int m2, int d2) {
   if (m1 < m2 - 1 || m1 > m2 + 1) {       // more than one month apart
      return true;
   } else if (m1 == m2 - 1 && d1 <= d2) {   // one month apart, days far
      return true;
   } else if (m1 == m2 + 1 && d1 >= d2) {   // one month apart, days far
```

```
        return true;
    } else {                          // same month
        return false;
    }
}
```

Exercise 5.18: digitSum
Write a method named digitSum that accepts an integer as a parameter and returns the sum of the digits of that number. For example, digitSum(29107) returns 2+9+1+0+7 or 19. For negative numbers, return the same value that would result if the number were positive. For example, digitSum(-456) returns 4+5+6 or 15.

```
public static int digitSum(int n) {
    n = Math.abs(n);          // handle negatives
    int sum = 0;
    while (n > 0) {
        sum = sum + (n % 10); // add last digit to sum
        n = n / 10;           // remove last digit
    }
    return sum;
}
```

Exercise 5.19: firstDigit
Write a method named firstDigit that returns the first digit of an integer. For example, firstDigit(3572) should return 3. It should work for negative numbers as well. For example, firstDigit(-947) should return 9.

Call	**Value Returned**
firstDigit(3572)	3
firstDigit(-947)	9
firstDigit(6)	6
firstDigit(35)	3
firstDigit(123456)	1

(Hint: Use a while loop. You may not use a String to solve this problem.)

```
public static int firstDigit(int num) {
    if (num < 0) {
        num = num * -1;
    }
    while (num >= 10) {
        num = num / 10;
    }
    return num;
}
```

Exercise 5.20: digitRange
Write a method named digitRange that accepts an integer as a parameter and returns the range of values of its digits. The range is defined as 1 more than the difference between the largest and smallest digit value. For example, the call of digitRange(68437) would return 6 because the largest digit value is 8 and the smallest is 3, so 8 - 3 + 1 = 6. If the number contains only one digit, return 1. You should solve this problem without using a String.

```
public static int digitRange(int n) {
    n = Math.abs(n);          // handle negatives
    int min = n % 10;
    int max = n % 10;
    while (n != 0) {
        int digit = n % 10;
        min = Math.min(min, digit);
        max = Math.max(max, digit);
        n = n / 10;
    }
    return max - min + 1;
}
```

Exercise 5.21: swapDigitOdd
Write a method named swapDigitPairs that accepts a positive integer *n* as a parameter and returns a new integer whose value is similar to *n*'s but with each pair of digits swapped in order. For example, the call of swapDigitPairs(482596) would return 845269. Notice that the 9 and 6 are swapped, as are the 2 and 5, and the 4 and 8. If the number contains an odd number of digits, leave the leftmost digit in its original place. For example, the call of swapDigitPairs(1234567) would return 1325476. You should solve this problem without using a String.

```
public static int swapDigitPairs(int n) {
    int result = 0;        // accumulate the result to return in here
    int power = 1;
    while (n / 10 != 0) {
        int right = n % 10;
        int left = n / 10 % 10;
        n = n / 100;
        result += left * power + right * power * 10;
        power *= 100;
    }
    result += n * power;   // leftmost odd remaining digit, if any
    return result;
}
```

Exercise 5.22: allDigitOdd
Write a method named allDigitsOdd that returns whether every digit of a given integer is odd. Your method should return true if the number consists entirely of odd digits and false if any of its digits are even. 0, 2, 4, 6, and 8 are even digits, and 1, 3, 5, 7, 9 are odd digits. Your method should work for positive and negative numbers.

For example, here are some calls to your method and their expected results:

Call	Value Returned
allDigitsOdd(4822116)	false
allDigitsOdd(135319)	true
allDigitsOdd(9175293)	false
allDigitsOdd(-137)	true

You should not use a String to solve this problem.

```
public static boolean allDigitsOdd(int n) {
    if (n == 0) {
        return false;
    }

    while (n != 0) {
        if (n % 2 == 0) {   // check whether last digit is odd
            return false;
        }
        n = n / 10;
    }
    return true;
}
```

Exercise 5.23: hasANOddDigit
Write a method named hasAnOddDigit that returns whether any digit of an integer is odd. Your method should return true if the number has at least one odd digit and false if none of its digits are odd. 0, 2, 4, 6, and 8 are even digits, and 1, 3, 5, 7, 9 are odd digits.

For example, here are some calls to your method and their expected results:

Call	Value Returned
hasAnOddDigit(4822116)	true
hasAnOddDigit(2448)	false
hasAnOddDigit(-7004)	true

You should not use a String to solve this problem.

```
public static boolean hasAnOddDigit(int n) {
   while (n != 0) {
      if (n % 2 != 0) {   // check whether last digit is odd
         return true;
      }
      n = n / 10;
   }
   return false;
}
```

Exercise 5.24: isAllVowels
Write a method named isAllVowels that returns whether a String consists entirely of vowels (a, e, i, o, or u, case-insensitively). If every character of the String is a vowel, your method should return true. If any character of the String is a non-vowel, your method should return false. Your method should return true if passed the empty string, since it does not contain any non-vowel characters.

For example, here are some calls to your method and their expected results:

Call	Value Returned
isAllVowels("eIEiO")	true
isAllVowels("oink")	false

```
public static boolean isAllVowels(String s) {
   for (int i = 0; i < s.length(); i++) {
      String letter = s.substring(i, i + 1);
      if (!isVowel(letter)) {
         return false;
      }
   }
   return true;
}

public static boolean isVowel(String s) {
   return s.equalsIgnoreCase("a") || s.equalsIgnoreCase("e") ||
   s.equalsIgnoreCase("i") || s.equalsIgnoreCase("o") ||
   s.equalsIgnoreCase("u");
}
```

Chapter 6: File Processing

Exercise 6.1: boyGirl

Write a method named boyGirl that accepts a Scanner as a parameter. The Scanner is reading its input from a file containing a series of names followed by integers. The names alternate between boys' names and girls' names. Your method should compute the absolute difference between the sum of the boys' integers and the sum of the girls' integers. The input could end with either a boy or girl; you may not assume that it contains an even number of names. If the input file tas.txt contains the following text:

```
Steve 3 Sylvia 7 Craig 14 Lisa 13 Brian 4 Charlotte 9 Jordan 6
```

then your method could be called in the following way:

```
Scanner input = new Scanner(new File("tas.txt"));
boyGirl(input);
```

and should produce the following output, since the boys' sum is 27 and the girls' sum is 29:

```
4 boys, 3 girls
Difference between boys' and girls' sums: 2
```

```
public static void boyGirl(Scanner input) {
    int boys = 0;
    int girls = 0;
    int boySum = 0;
    int girlSum = 0;
    while (input.hasNext()) {
        String throwAway = input.next();  // throw away name
        if (boys == girls) {
            boys++;
            boySum += input.nextInt();
        } else {
            girls++;
            girlSum += input.nextInt();
        }
    }
    System.out.println(boys + " boys, " + girls + " girls");
    System.out.println("Difference between boys' and girls' sums: " + Math.abs(boySum - girlSum
));
}
```

Exercise 6.2: evenNumbers

Write a method named evenNumbers that accepts a Scanner as a parameter reading input from a file containing a series of integers, and report various statistics about the integers. You may assume that there is at least one integer in the file. Report the total number of numbers, the sum of the numbers, the count of even numbers and the percent of even numbers. For example, if a Scannerinput on file numbers.txt contains the following text:

```
5 7 2 8 9 10 12 98 7 14 20 22
```

then the call evenNumbers(input); should produce the following output:

```
12 numbers, sum = 214
8 evens (66.67%)
```

```
public static void evenNumbers(Scanner input) {
   int count = 0;
   int evens = 0;
   int sum = 0;
   while (input.hasNextInt()) {
      int number = input.nextInt();
      count++;
      sum += number;
      if (number % 2 == 0) {
         evens++;
      }
   }
   double percent = 100.0 * evens / count;
   System.out.println(count + " numbers, sum = " + sum);
   System.out.printf("%d evens (%.2f%%)\n", evens, percent);
}
```

Exercise 6.3: negativeSum

Write a method named negativeSum that accepts a Scanner as a parameter reading input from a file containing a series of integers, and determine whether the sum starting from the first number is ever negative. The method should print a message indicating whether a negative sum is possible and should return true if a negative sum can be reached and false if not. For example, if the file contains the following text, your method will consider the sum of just one number (38), the sum of the first two numbers (38 + 4), the sum of the first three numbers (38 + 4 + 19), and so on up to the sum of all of the numbers:

```
38 4 19 -27 -15 -3 4 19 38
```

None of these sums is negative, so the method would produce the following message and return false:

```
no negative sum
```

If the file instead contains the following numbers, the method finds that a negative sum of -8 is reached after adding 6 numbers together (14 + 7 + -10 + 9 + -18 + -10):

```
14 7 -10 9 -18 -10 17 42 98
```

It should output the following and return true, indicating that a negative sum can be reached:

```
-8 after 6 steps
```

```
public static boolean negativeSum(Scanner input) {
    int sum = 0;
    int count = 0;
    while (input.hasNextInt()) {
        int next = input.nextInt();
        sum += next;
        count++;
        if (sum < 0) {
            System.out.println(sum + " after " + count + " steps");
            return true;
        }
    }
    System.out.println("no negative sum");
    return false;  // not found
}
```

Exercise 6.4: countCoins
Marty Stepp (on 2016/09/08)

Write a method named countCoins that accepts as its parameter a Scanner for an input file whose data represents a person's money grouped into stacks of coins. Your method should add up the cash values of all the coins and print the total money at the end. The input consists of a series of pairs of tokens, where each pair begins with an integer and is followed by the type of coin, which will be either "pennies" (1 cent each), "nickels" (5 cents each), "dimes" (10 cents each), or "quarters" (25 cents each), case-insensitively. A given coin might appear more than once on the same line.

For example, if the input file contains the following text:

```
3 pennies 2 quarters 1 pennies 3 nickels 4 dimes
```

3 pennies are worth 3 cents, and 2 quarters are worth 50 cents, and 1 penny is worth 1 cent, and 3 nickels are worth 15 cents, and 4 dimes are worth 40 cents. The total of these is 1 dollar and 9 cents, therefore your method would produce the following output if passed this input data. Notice that the method should show exactly two digits after the decimal point, so it says 09 for 9 cents:

```
Total money: $1.09
```

Here is a second example. Suppose the input file contains the following text. Notice the capitalization and spacing:

```
12   QUARTERS     1   Pennies     33
PeNnIeS

  10    niCKELs
```

Then your method would produce the following output:

```
Total money: $3.84
```

You may assume that the file contains at least 1 pair of tokens. You may also assume that the input is valid; that the input has an even number of tokens, that every other token is an integer, and that the others are valid coin types.

```
// count up money as a double of dollars/cents; use printf at end
public static void countCoins(Scanner input) {
    double total = 0.0;
    while (input.hasNext()) {
        int count = input.nextInt();
        String coin = input.next().toLowerCase();
        if (coin.equals("nickels")) {
            count = count * 5;
        } else if (coin.equals("dimes")) {
            count = count * 10;
        } else if (coin.equals("quarters")) {
            count = count * 25;
        }
        total = total + (double) count / 100;
    }
    System.out.printf("Total money: $%.2f\n", total);
}
```

Exercise 6.5: collapseSpaces

Write a static method named collapseSpaces that accepts a Scanner representing a file as a parameter and writes that file's text to the console, with multiple spaces or tabs reduced to single spaces between words that appear on the same line. For example, if a Scanner variable named input is reading an input file containing the following text:

```
four      score   and

seven              years ago         our

fathers brought            forth
   on this          continent
a         new

nation
```

then the call collapseSpaces(input); should produce the following output:

```
four score and

seven years ago our

fathers brought forth
on this continent
a new

nation
```

Each word is to appear on the same line in output as it appears in the file. Notice that lines can be blank.

```
public static void collapseSpaces(Scanner input) {
    while (input.hasNextLine()) {
        String text = input.nextLine();
        Scanner words = new Scanner(text);
        if (words.hasNext()) {
            String word = words.next();
            System.out.print(word);
            while (words.hasNext()) {
                word = words.next();
                System.out.print(" " + word);
            }
        }
        System.out.println();
    }
}
```

Exercise 6.6: readEntireFile
Write a method named readEntireFile that accepts a Scanner representing an input file as its parameter, then reads that file and returns the entire text contents of that file as a String.

```
public static String readEntireFile(Scanner input) {
    String text = "";
    while (input.hasNextLine()) {
        text += input.nextLine() + "\n";
    }
    return text;
}
```

Exercise 6.7: flipLines
Write a method named flipLines that accepts as its parameter a Scanner for an input file and that writes to the console the same file's contents with successive pairs of lines reversed in order. For example, if the input file contains the following text:

```
Twas brillig and the slithy toves
did gyre and gimble in the wabe.
All mimsey were the borogroves,
and the mome raths outgrabe.

"Beware the Jabberwock, my son,
the jaws that bite, the claws that catch,
Beware the JubJub bird and shun
the frumious bandersnatch."
```

The program should print the first pair of lines in reverse order, then the second pair in reverse order, then the third pair in reverse order, and so on. Therefore your method should produce the following output to the console:

```
did gyre and gimble in the wabe.
Twas brillig and the slithy toves
and the mome raths outgrabe.
All mimsey were the borogroves,
"Beware the Jabberwock, my son,

Beware the JubJub bird and shun
the jaws that bite, the claws that catch,
the frumious bandersnatch."
```

Notice that a line can be blank, as in the third pair. Also notice that an input file can have an odd number of lines, as in the one above, in which case the last line is printed in its original position. You may not make any assumptions about how many lines are in the Scanner.

```
public static void flipLines(Scanner input) {
   while (input.hasNextLine()) {
      String first = input.nextLine();
      if (input.hasNextLine()) {
         String second = input.nextLine();
         System.out.println(second);
      }
      System.out.println(first);
   }
}
```

Exercise 6.8: doubleSpace
Write a method named doubleSpace that accepts a Scanner for an input file and a PrintStream for an output file as its parameters, writing into the output file a double-spaced version of the text in the input file. You can achieve this task by inserting a blank line between each line of output.

```
public static void doubleSpace(Scanner in, PrintStream out) {
   while (in.hasNextLine()) {
      out.println(in.nextLine());
      out.println();
   }
   out.close();
}
```

Exercise 6.9: wordWrap
Write a method called wordWrap that accepts a Scanner representing an input file as its parameter and outputs each line of the file to the console, word-wrapping all lines that are longer than 60 characters. For example, if a line contains 112 characters, the method should replace it with two lines: one containing the first 60 characters and another containing the final 52

characters. A line containing 217 characters should be wrapped into four lines: three of length 60 and a final line of length 37.

```
public static void wordWrap(Scanner input) {
   while (input.hasNextLine()) {
      String line = input.nextLine();
      while (line.length() > 60) {
         String first60 = line.substring(0, 60);
         System.out.println(first60);
         line = line.substring(60);
      }
      System.out.println(line);
   }
}
```

Exercise 6.10: wordWrap2
Modify the preceding wordWrap method into a new wordWrap2 method that accepts a second parameter for a PrintStream to write the data, and outputs the newly wrapped text to that PrintStream rather than to the console. Also, modify it to use a local variable to store the maximum line length rather than hard-coding 60. (You can go back to the last problem to copy your solution code from there and paste it here as a starting point.)

```
public static void wordWrap2(Scanner input, PrintStream out) {
   int max = 60;
   while (input.hasNextLine()) {
      String line = input.nextLine();
      while (line.length() > max) {
         String first = line.substring(0, max);
         out.println(first);
         line = line.substring(max);
      }
      out.println(line);
   }
   out.close();
}
```

Exercise 6.11: wordWrap3
Modify the preceding wordWrap method into a new wordWrap3 method that wraps only whole words, never chopping a word in half. Assume that a word is any whitespace-separated token and that all words are under 60 characters in length. Make sure that each time you wrap a line, the subsequent wrapped line(s) each begin with a word and not with any leading whitespace. Accept only a single parameter for the input Scanner, and send your method's output to the console, as in the original wordWrap problem; do not use an output file as was done in wordWrap2.

```
public static void wordWrap3(Scanner input) throws FileNotFoundException {
   int max = 60;
```

```
    while (input.hasNextLine()) {
        String line = input.nextLine();
        while (line.length() > max) {
            // find the nearest token boundary
            int index = max;
            while (!Character.isWhitespace(line.charAt(index))) {
                index--;
            }
            String first = line.substring(0, index + 1);
            System.out.println(first);
            line = line.substring(index + 1);
        }
        System.out.println(line);
    }
}
```

Exercise 6.12: stripHtmlTags

Write a method called stripHtmlTags that accepts a Scanner representing an input file containing an HTML web page as its parameter, then reads that file and prints the file's text with all HTML tags removed. A tag is any text between the characters < and > . For example, consider the following text:

```
<html>
<head>
<title>My web page</title>
</head>
<body>
<p>There are many pictures of my cat here,
as well as my <b>very cool</b> blog page,
which contains <font color="red">awesome
stuff about my trip to Vegas.</p>

Here's my cat now:<img src="cat.jpg">
</body>
</html>
```

If the file contained these lines, your program should output the following text:

```



My web page



There are many pictures of my cat here,
as well as my very cool blog page,
which contains awesome
stuff about my trip to Vegas.

Here's my cat now:
```

You may assume that the file is a well-formed HTML document and that there are no < or > characters inside tags.

```java
public static void stripHtmlTags(Scanner input) throws FileNotFoundException {
   String text = "";
   while (input.hasNextLine()) {
      text += input.nextLine() + "\n";
   }
   int indexOfTag = text.indexOf("<");
   while (indexOfTag >= 0) {
      String start = text.substring(0, indexOfTag);
      text = text.substring(indexOfTag, text.length());
      int indexOfTagEnd = text.indexOf(">");
      text = start + text.substring(indexOfTagEnd + 1, text.length());
      indexOfTag = text.indexOf("<");
   }
   System.out.print(text);
}
```

Exercise 6.13: stripComents

Write a method called stripComments that accepts a Scanner representing an input file containing a Java program as its parameter, reads that file, and then prints the file's text with all comments removed. A comment is any text on a line from // to the end of the line, and any text between /* and */ characters. For example, consider the following text:

```java
import java.util.*;
/* My program
by Suzy Student */
public class Program {
   public static void main(String[] args) {
      System.out.println("Hello, world!"); // a println
   }

   public static /* Hello there */ void foo() {
      System.out.println("Goodbye!"); // comment here
   } /* */
}
```

If the file contained this text, your program should output the following text:

```
import java.util.*;

public class Program {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }

   public static  void foo() {
      System.out.println("Goodbye!");
   }
}

public static void stripComments(Scanner input) throws FileNotFoundException {
   String text = "";
   while (input.hasNextLine()) {
      text += input.nextLine() + "\n";
   }
   int index1 = text.indexOf("//");
   int index2 = text.indexOf("/*");
   while (index1 >= 0 || index2 >= 0) {
      if (index2 < 0 || (index1 >= 0 && index1 < index2)) {
         String start = text.substring(0, index1);
         text = text.substring(index1, text.length());
         text = start + text.substring(text.indexOf("\n"), text.length());
      } else {
         String start = text.substring(0, index2);
         text = text.substring(index2, text.length());
         text = start + text.substring(text.indexOf("*/") + 2, text.length());
      }
      index1 = text.indexOf("//");
      index2 = text.indexOf("/*");
   }
   System.out.print(text);
}
```

Exercise 6.14: printDuplicates

Write a method named printDuplicates that accepts as its parameter a Scanner for an input file containing a series of lines. Your method should examine each line looking for consecutive occurrences of the same token on the same line and print each duplicated token along how many times it appears consecutively. Non-repeated tokens are not printed. Repetition across multiple lines (such as if a line ends with a given token and the next line starts with the same token) is not considered in this problem.

For example, if the input file contains the following text:

```
hello how how are you you you you
I I I am Jack's Jack's smirking smirking smirking smirking smirking revenge
   bow  wow wow yippee yippee   yo yippee   yippee yay  yay yay
one fish two fish red fish blue fish
It's the Muppet Show, wakka wakka wakka
```

Your method would produce the following output for the preceding input file:

```
how*2 you*4
I*3 Jack's*2 smirking*5
wow*2 yippee*2 yippee*2 yay*3

wakka*3
```

Your code prints only the repeated tokens; the ones that only appear once in a row are not shown. Your code should place a single space between each reported duplicate token and should respect the line breaks in the original file. This is why a blank line appears in the expected output, corresponding to the fourth line of the file that did not contain any consecutively duplicated tokens. You may assume that each line of the file contains at least 1 token of input.

```
public static void printDuplicates(Scanner input) {
    while (input.hasNextLine()) {
        String line = input.nextLine();
        Scanner lineScan = new Scanner(line);
        String token = lineScan.next();
        int count = 1;
        while (lineScan.hasNext()) {
            String token2 = lineScan.next();
            if (token2.equals(token)) {
                count++;
            } else {
                if (count > 1) {
                    System.out.print(token + "*" + count + " ");
                }
                token = token2;
                count = 1;
            }
        }
        if (count > 1) {
            System.out.print(token + "*" + count);
        }
        System.out.println();
    }
```

}

Exercise 6.15: coinflip
Write a method named coinFlip that accepts as its parameter a Scanner for an input file. Assume that the input file data represents results of sets of coin flips that are either heads (H) or tails (T) in either upper or lower case, separated by at least one space. Your method should consider each line to be a separate set of coin flips and should output to the console the number of heads and the percentage of heads in that line, rounded to the nearest tenth. If this percentage is more than 50%, you should print a "You win" message. For example, consider the following input file:

```
H T H H T
T t  t T h  H
  h
```

For the input above, your method should produce the following output:

```
3 heads (60.0%)
You win!

2 heads (33.3%)

1 heads (100.0%)
You win!
```

The format of your output must exactly match that shown above. You may assume that the Scanner contains at least 1 line of input, that each line contains at least one token, and that no tokens other than h, H, t, or T will be in the lines.

```
public static void coinFlip(Scanner input) {
    while (input.hasNextLine()) {
        Scanner lineScan = new Scanner(input.nextLine().toUpperCase());
        int heads = 0;
        int total = 0;
        while (lineScan.hasNext()) {
            total++;
            if (lineScan.next().equals("H")) {
                heads++;
            }
        }

        double percent = 100.0 * heads / total;
        System.out.printf("%d heads (%.1f%%)\n", heads, percent);
        if (percent > 50) {
            System.out.println("You win!");
        }
        System.out.println();
    }
}
```

Exercise 6.16: mostCommonNames
Write a method named mostCommonNames that accepts as its parameter a Scanner for an input file whose data is a sequence of lines, where each line contains a set of first names separated by spaces. Your method should print the name that occurs the most frequently in each line of the file. The method should also return the total number of unique names that were seen in the file. You may assume that no name appears on more than one line of the file.

Each line should be considered separately from the others. On a given line, some names are repeated; all occurrences of a given name will appear consecutively in the file. If two or more names occur the same number of times, print the one that appears earlier in the file. If every single name on a given line is different, every name will have 1 occurrence, so you should just print the first name in the file.

For example, if the input file contains the following text:

```
Benson Eric   Eric  Marty Kim  Kim Kim   Jenny  Nancy Nancy  Nancy  Paul  Paul
Stuart Stuart Stuart Ethan Alyssa Alyssa Helene Jessica Jessica Jessica Jessica
Jared  Alisa Yuki   Catriona  Cody   Coral   Trent Kevin  Ben Stefanie Kenneth
```

On the first line, there is one occurrence of the name Benson, two occurrences of Eric, one occurrence of Marty, three occurrences of Kim, one of Jenny, three of Nancy, and two of Paul. Kim and Nancy appear the most times (3), and Kim appears first in the file. So for that line, your method should print that the most common is Kim. The complete output would be the following. The method would also return 23, since there are 23 unique names in the entire file.

```
Most common: Kim
Most common: Jessica
Most common: Jared
```

You may assume that there is at least one line of data in the file and that each line will contain at least one name.

```
public static int mostCommonNames(Scanner input) {
    int totalCount = 0;
    while (input.hasNextLine()) {
        String line = input.nextLine();
        Scanner words = new Scanner(line);

        String mostCommon = words.next();
        int mostCount = 1;
        String current = mostCommon;
        int currentCount = 1;
        totalCount++;

        while (words.hasNext()) {
            String next = words.next();
            if (next.equals(current)) {
                currentCount++;
                if (currentCount > mostCount) {
                    mostCount = currentCount;
                    mostCommon = current;
                }
            } else {
                current = next;
                currentCount = 1;
                totalCount++;
            }
        }
        System.out.println("Most common: " + mostCommon);
    }
    return totalCount;
}
```

Exercise 6.17: inoutStats

Write a method named inputStats that takes a Scanner representing a file as a parameter and that reports various statistics about the file's text. In particular, your method should report the number of lines in the file, the longest line, the number of tokens on each line, and the length of the longest token on each line. You may assume that the input file has at least one line of input and that each line has at least one token. For example, if a Scanner named input on file carroll.txt contains the following text:

```
"Beware the Jabberwock, my son,
the jaws that bite, the claws that catch,
Beware the JubJub bird and shun
the frumious bandersnatch."
```

then the call inputStats(input); should produce the following output:

```
Line 1 has 5 tokens (longest = 11)
Line 2 has 8 tokens (longest = 6)
Line 3 has 6 tokens (longest = 6)
Line 4 has 3 tokens (longest = 14)
Longest line: the jaws that bite, the claws that catch,
```

```
public static void inputStats(Scanner input) {
   int lines = 0;
   String longestLine = "";

   while (input.hasNextLine()) {
      String line = input.nextLine();
      lines++;
      if (line.length() > longestLine.length()) {
         longestLine = line;
      }

      Scanner lineScan = new Scanner(line);
      int tokens = 0;
      int longest = 0;
      while (lineScan.hasNext()) {
         String token = lineScan.next();
         tokens++;
         longest = Math.max(longest, token.length());
      }

      System.out.println("Line " + lines + " has " + tokens +
      " tokens (longest = " + longest + ")");
   }
   System.out.println("Longest line: " + longestLine);
}
```

Exercise 6.18: plusScores

Write a static method named plusScores that accepts as a parameter a Scanner containing a series of lines that represent student records. Each student record takes up two lines of input. The first line has the student's name and the second line has a series of plus and minus characters. Below is a sample input:

```
Kane, Erica
--+-+
Chandler, Adam
```

```
++-+
Martin, Jake
+++++++
Dillon, Amanda
++-++-+-
```

The number of plus/minus characters will vary, but you may assume that at least one such character appears and that no other characters appear on the second line of each pair. For each student you should produce a line of output withthe student's name followed by a colon followed by the percent of plus characters. For example, if the input above is stored in a Scanner called input, the call of plusScores(input); should produce the following output:

```
Kane, Erica: 40.0% plus
Chandler, Adam: 75.0% plus
Martin, Jake: 100.0% plus
Dillon, Amanda: 62.5% plus
```

```
public static void plusScores(Scanner input) {
    while (input.hasNextLine()) {
        String name = input.nextLine();
        String data = input.nextLine();
        int plus = 0;
        int count = 0;
        for (int i = 0; i < data.length(); i++) {
            count++;
            if (data.charAt(i) == '+') {
                plus++;
            }
        }
        double percent = 100.0 * plus / count;
        System.out.println(name + ": " + percent + "% plus");
    }
}
```

Exercise 6.19: leetSpeak

Write a method leetSpeak that accepts two parameters: a Scanner representing an input file, and a PrintStream representing an output file. Your method should convert the input file's text to "leet speak" (aka 1337 speak), an internet dialect where various letters are replaced by other letters/numbers. Output the leet version of the text to the given output file. Preserve the original line breaks from the input. Also wrap each word of input in parentheses. Perform the following replacements:

Original character	'Leet' character
o	0
l (lowercase L)	1
e	3
a	4
t	7
s (at the end of a word only)	Z

For example, if the input file lincoln.txt contains the following text:

```
four score and
seven years ago our

fathers brought forth on this continent
a new nation
```

And your method is called in the following way:

```java
Scanner input = new Scanner(new File("lincoln.txt"));
PrintStream output = new PrintStream(new File("leet.txt"));
leetSpeak(input, output);
```

Then after the call, the output file leet.txt should contain the following text:

```
(f0ur) (sc0r3) (4nd)
(s3v3n) (y34rZ) (4g0) (0ur)

(f47h3rZ) (br0ugh7) (f0r7h) (0n) (7hiZ) (c0n7in3n7)
(4) (n3w) (n47i0n)
```

You may assume that each token from the input file is separated by exactly one space.

Hint: You may want to use the String object's replace method, which is used as follows:

```java
String str = "mississippi";
str = str.replace("s", "*");    // str = "mi**i**ippi"
```

```java
public static void leetSpeak(Scanner input, PrintStream output) {
   while (input.hasNextLine()) {
      String line = input.nextLine();
```

```
        Scanner lineScanner = new Scanner(line);
        while (lineScanner.hasNext()) {
            String word = lineScanner.next();
            word = word.replace("o", "0");
            word = word.replace("l", "1");
            word = word.replace("e", "3");
            word = word.replace("a", "4");
            word = word.replace("t", "7");
            if (word.endsWith("s")) {
                word = word.substring(0, word.length() - 1) + "Z";
            }
            output.print("(" + word + ") ");
        }
        output.println();
    }
}
```

Chapter 7: Arrays

Exercise 7.1: lastIndexOf

Write a method named lastIndexOf that accepts an array of integers and an integer value as its parameters and returns the last index at which the value occurs in the array. The method should return -1 if the value is not found. For example, in the list containing {74, 85, 102, 99, 101, 85, 56}, the last index of the value 85 is 5.

```
public static int lastIndexOf(int[] a, int target) {
    for (int i = a.length - 1; i >= 0; i--) {
        if (a[i] == target) {
            return i;
        }
    }
    return -1;
}
```

Exercise 7.2: range

Write a static method named range that takes an array of integers as a parameter and returns the range of values contained in the array. The range of an array is defined to be one more than the difference between its largest and smallest element. For example, if the largest element in the array is 15 and the smallest is 4, the range is 12. If the largest and smallest values are the same, the range is 1.

The following table shows some calls to your method and their results (the largest and smallest values are underlined):

Call	Value Returned
int[] a1 = {8, 3, 5, 7, 2, 4};	range(a1) returns 7
int[] a2 = {15, 22, 8, 19, 31};	range(a2) returns 24
int[] a3 = {3, 10000000, 5, -29, 4};	range(a3) returns 10000030
int[] a4 = {100, 5};	range(a4) returns 96
int[] a5 = {32};	range(a5) returns 1

You may assume that the array contains at least one element (that its length is at least 1). You should not make any assumptions about the values of the particular elements in the array; they could be extremely large, very small, etc. You should not modify the contents of the array.

```
public static int range(int[] a) {
    int min = 0;
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        if (i == 0 || a[i] < min) {
            min = a[i];
        }
        if (i == 0 || a[i] > max) {
            max = a[i];
        }
    }
    int valueRange = max - min + 1;
    return valueRange;
}
```

Exercise 7.3: countInRange

Write a method called countInRange that accepts an array of integers, a minimum value, and a maximum value as parameters and returns the count of how many elements from the array fall between the minimum and maximum (inclusive).

For example, in the array {14, 1, 22, 17, 36, 7, -43, 5}, there are four elements whose values fall between 4 and 17.

```
public static int countInRange(int[] a, int min, int max) {
    int count = 0;
    for (int i = 0; i < a.length; i++) {
        if (a[i] >= min && a[i] <= max) {
            count++;
        }
    }
    return count;
}
```

Exercise 7.4: isSorted

Write a static method named isSorted that accepts an array of doubles as a parameter and returns true if the list is in sorted (nondecreasing) order and false otherwise. For example, if arrays named list1 and list2 store {16.1, 12.3, 22.2, 14.4} and {1.5, 4.3, 7.0, 19.5, 25.1, 46.2} respectively, the calls isSorted(list1) and isSorted(list2) should return false and true respectively. Assume the array has at least one element. A one-element array is considered to be sorted.

```
public static boolean isSorted(double[] list) {
    for (int i = 0; i < list.length - 1; i++) {
        if (list[i] > list[i + 1]) {
            return false;
        }
    }
    return true;
```

}

Exercise 7.5: mode

Write a method called mode that returns the most frequently occurring element of an array of integers. Assume that the array has at least one element and that every element in the array has a value between 0 and 100 inclusive. Break ties by choosing the lower value.

For example, if the array passed contains the values {27, 15, 15, 11, 27}, your method should return 15. (*Hint:* You may wish to look at the Tally program from earlier in this chapter to get an idea of how to solve this problem.)

```
public static int mode(int[] a) {
   // tally all the occurrences of each element
   int[] tally = new int[101];
   for (int i = 0; i < a.length; i++) {
      tally[a[i]]++;
   }
   // scan the array of tallies to find the highest tally (the mode)
   int maxCount = 0;
   int modeValue = 0;
   for (int i = 0; i < tally.length; i++) {
      if (tally[i] > maxCount) {
         maxCount = tally[i];
         modeValue = i;
      }
   }
   return modeValue;
}
```

Exercise 7.6: stdev

Write a method called stdev that returns the standard deviation of an array of integers. Standard deviation is computed by taking the square root of the sum of the squares of the differences between each element and the mean, divided by one less than the number of elements. (It's just that simple!)

More concisely and mathematically, the standard deviation of an array a is written as follows:

$$stdev(a) = \sqrt{\frac{\sum_{i=0}^{a.length-1}(a[i] - average(a))^2}{a.length - 1}}$$

For example, if the array passed contains the values {1, -2, 4, -4, 9, -6, 16, -8, 25, -10}, your method should return approximately 11.237. You may assume that the array passed is non-null and contains at least two values, because the standard deviation is undefined otherwise.

(Note: you might fail the last two tests because of rounding, but as long as it's close, then your algorithm is probably correct.)

```
public static double stdev(int[] a) {
   if (a.length == 1) {
      return 0.0;
   }
   int sum = 0;
   for (int i = 0; i < a.length; i++) {
      sum += a[i];
   }
   double average = (double) sum / a.length;
   double sumDiff = 0.0;
   for (int i = 0; i < a.length; i++) {
      sumDiff += Math.pow(a[i] - average, 2);
   }
   return Math.abs(Math.sqrt(sumDiff / (a.length - 1)));
}
```

Exercise 7.7: kthLargest

Write a method called kthLargest that accepts an integer k and an array a as its parameters and returns the element such that *k* elements have greater or equal value. If k = 0, return the largest element; if k = 1, return the second largest element, and so on.

For example, if the array passed contains the values {74, 85, 102, 99, 101, 56, 84} and the integer *k* passed is 2, your method should return 99 because there are two values at least as large as 99 (101 and 102).

Assume that 0 <= *k* < *a*.length. (Hint: Consider sorting the array, or a copy of the array first.)

```
public static int kthLargest(int k, int[] a) {
    int[] a2 = new int[a.length];
    for (int i = 0; i < a.length; i++) {
        a2[i] = a[i];
    }
    Arrays.sort(a2);
    return a2[a2.length - 1 - k];
}
```

Exercise 7.8: median

Write a method called median that accepts an array of integers as its argument and returns the median of the numbers in the array. The median is the number that will appear in the middle if you arrange the elements in order. Assume that the array is of odd size (so that one sole element constitutes the median) and that the numbers in the array are between 0 and 99 inclusive.

For example, the median of {5, 2, 4, 17, 55, 4, 3, 26, 18, 2, 17} is 5, and the median of {42, 37, 1, 97, 1, 2, 7, 42, 3, 25, 89, 15, 10, 29, 27} is 25.

(*Hint*: You may wish to look at the Tally program from earlier in this chapter for ideas.)

```
public static int median(int[] a) {
    // count the number of occurrences of each number into a "tally" array
    int[] tally = new int[100];
    for (int i = 0; i < a.length; i++) {
        tally[a[i]]++;
    }
    // examine the tallies and stop when we have seen half the numbers
    int i;
    for (i = 0; tally[i] <= a.length / 2; i++) {
        tally[i + 1] += tally[i];
    }
    return i;
}
```

Exercise 7.9: minGap

Write a method named minGap that accepts an integer array as a parameter and returns the minimum 'gap' between adjacent values in the array. The gap between two adjacent values in a array is defined as the second value minus the first value. For example, suppose a variable called array is an array of integers that stores the following sequence of values.

```
int[] array = {1, 3, 6, 7, 12};
```

The first gap is 2 (3 - 1), the second gap is 3 (6 - 3), the third gap is 1 (7 - 6) and the fourth gap is 5 (12 - 7). Thus, the call of minGap(array) should return 1 because that is the smallest gap in the array. Notice that the minimum gap could be a negative number. For example, if array stores the following sequence of values:

{3, 5, 11, 4, 8}

The gaps would be computed as 2 (5 - 3), 6 (11 - 5), -7 (4 - 11), and 4 (8 - 4). Of these values, -7 is the smallest, so it would be returned.

This gap information can be helpful for determining other properties of the array. For example, if the minimum gap is greater than or equal to 0, then you know the array is in sorted (nondecreasing) order. If the gap is greater than 0, then you know the array is both sorted and unique (strictly increasing).

If you are passed an array with fewer than 2 elements, you should return 0.

```
public static int minGap(int[] list) {
   if (list.length < 2) {
      return 0;
   } else {
      int min = list[1] - list[0];
      for (int i = 2; i < list.length; i++) {
         int gap = list[i] - list[i - 1];
         if (gap < min) {
            min = gap;
         }
      }
      return min;
   }
}
```

Exercise 7.10: percentEven
Write a method called percentEven that accepts an array of integers as a parameter and returns the percentage of even numbers in the array as a real number. For example, if the array stores the elements {6, 2, 9, 11, 3}, then your method should return 40.0. If the array contains no even elements or no elements at all, return 0.0.

```
public static double percentEven(int[] list) {
   if (list.length == 0) {
      return 0.0;
   }
   int numEven = 0;
   for (int element: list) {
      if (element % 2 == 0) {
         numEven++;
      }
   }
   return numEven * 100.0 / list.length;
}
```

Exercise 7.11: isUnique
Whitaker Brand (on 2016/09/08)

Write a method named isUnique that takes an array of integers as a parameter and that returns a boolean value indicating whether or not the values in the array are unique (true for yes, false for no). The values in the list are considered unique if there is no pair of values that are equal. For example, if a variable called list stores the following values:

```
int[] list = {3, 8, 12, 2, 9, 17, 43, -8, 46, 203, 14, 97, 10, 4};
```

Then the call of isUnique(list) should return true because there are no duplicated values in this list. If instead the list stored these values:

```
int[] list = {4, 7, 2, 3, 9, 12, -47, -19, 308, 3, 74};
```

Then the call should return false because the value 3 appears twice in this list. Notice that given this definition, a list of 0 or 1 elements would be considered unique.

```
public static boolean isUnique(int[] list) {
   for (int i = 0; i < list.length; i++) {
      for (int j = i + 1; j < list.length; j++) {
         if (list[i] == list[j]) {
            return false;
         }
      }
   }
   return true;
}
```

Exercise 7.12: priceIsRight

Write a method priceIsRight that accepts an array of integers *bids* and an integer *price* as parameters. The method returns the element in the *bids* array that is closest in value to *price* without being larger than *price*. For example, if *bids* stores the elements {200, 300, 250, 999, 40}, then priceIsRight(bids, 280) should return 250, since 250 is the bid closest to 280 without going over 280. If all bids are larger than *price*, then your method should return -1.

The following table shows some calls to your method and their expected results:

Arrays	Returned Value
int[] a1 = {900, 885, 989, 1};	priceIsRight(a1, 800) returns 1
int[] a2 = {200};	priceIsRight(a2, 120) returns -1
int[] a3 = {500, 300, 241, 99, 501};	priceIsRight(a3, 50) returns -1

You may assume there is at least 1 element in the array, and you may assume that the *price* and the values in *bids* will all be greater than or equal to 1. Do not modify the contents of the array passed to your method as a parameter.

```
public static int priceIsRight(int[] bids, int price) {
   int bestPrice = -1;
   for (int i = 0; i < bids.length; i++) {
      if (bids[i] <= price && bids[i] > bestPrice) {
         bestPrice = bids[i];
      }
   }
   return bestPrice;
}
```

Exercise 7.13: longestSortedSequence
Write a method named longestSortedSequence that accepts an array of integers as a parameter and that returns the length of the longest sorted (nondecreasing) sequence of integers in the array. For example, if a variable named array stores the following values:

```
int[] array = {3, 8, 10, 1, 9, 14, -3, 0, 14, 207, 56, 98, 12};
```

then the call of longestSortedSequence(array) should return 4 because the longest sorted sequence in the array has four values in it (the sequence -3, 0, 14, 207). Notice that sorted means nondecreasing, which means that the sequence could contain duplicates. For example, if the array stores the following values:

```
int[] array2 = {17, 42, 3, 5, 5, 5, 8, 2, 4, 6, 1, 19}
```

Then the method would return 5 for the length of the longest sequence (the sequence 3, 5, 5, 5, 8). Your method should return 0 if passed an empty array.

```
public static int longestSortedSequence(int[] list) {
   if (list.length == 0) {
       return 0;
   }
   int max = 1;
   int count = 1;
   for (int i = 1; i < list.length; i++) {
       if (list[i] >= list[i - 1]) {
           count++;
       } else {
           count = 1;
       }
       if (count > max) {
           max = count;
       }
   }
   return max;
}
```

Exercise 7.14: contains

Write a static method named contains that accepts two arrays of integers *a1* and *a2* as parameters and that returns a boolean value indicating whether or not *a2*'s sequence of elements appears in *a1* (true for yes, false for no). The sequence of elements in *a2* may appear anywhere in *a1* but must appear consecutively and in the same order. For example, if variables called list1 and list2 store the following values:

```
int[] list1 = {1, 6, 2, 1, 4, 1, 2, 1, 8};
int[] list2 = {1, 2, 1};
```

Then the call of contains(list1, list2) should return true because list2's sequence of values {1, 2, 1} is contained in list1 starting at index 5. If list2 had stored the values {2, 1, 2}, the call of contains(list1, list2) would return false because list1 does not contain that sequence of values. Any two lists with identical elements are considered to contain each other, so a call such as contains(list1, list1) should return true.

You may assume that both arrays passed to your method will have lengths of at least 1. You may not use any Strings to help you solve this problem, nor methods that produce Strings such as Arrays.toString.

```
public static boolean contains(int[] a1, int[] a2) {
    for (int i = 0; i <= a1.length - a2.length; i++) {
        boolean found = true;
        for (int j = 0; j < a2.length; j++) {
            if (a1[i + j] != a2[j]) {
```

```
            found = false;
         }
      }
      if (found) {
         return true;
      }
   }
   return false;
}
```

Exercise 7.15: collapse
Write a method called collapse that accepts an array of integers as a parameter and returns a new array containing the result of replacing each pair of integers with the sum of that pair. For example, if an array called list stores the values {7, 2, 8, 9, 4, 13, 7, 1, 9, 10}, then the call of collapse(list) should return a new array containing {9, 17, 17, 8, 19}. The first pair from the original list is collapsed into 9 (7 + 2), the second pair is collapsed into 17 (8 + 9), and so on. If the list stores an odd number of elements, the final element is not collapsed. For example, if the list had been {1, 2, 3, 4, 5}, then the call would return {3, 7, 5}. Your method should not change the array that is passed as a parameter.

```
public static int[] collapse(int[] list) {
   int[] result = new int[list.length / 2 + list.length % 2];
   for (int i = 0; i < result.length - list.length % 2; i++) {
      result[i] = list[2 * i] + list[2 * i + 1];
   }
   if (list.length % 2 == 1) {
      result[result.length - 1] = list[list.length - 1];
   }
   return result;
}
```

Exercise 7.16: append
Write a method called append that accepts two integer arrays as parameters and returns a new array that contains the result of appending the second array's values at the end of the first array. For example, if arrays list1 and list2 store {2, 4, 6} and {1, 2, 3, 4, 5} respectively, the call of append(list1, list2) should return a new array containing {2, 4, 6, 1, 2, 3, 4, 5}. If the call instead had been append(list2, list1), the method would return an array containing {1, 2, 3, 4, 5, 2, 4, 6}.

```
public static int[] append(int[] list1, int[] list2) {
   int[] result = new int[list1.length + list2.length];
   for (int i = 0; i < list1.length; i++) {
      result[i] = list1[i];
   }
   for (int i = 0; i < list2.length; i++) {
      result[i + list1.length] = list2[i];
   }
   return result;
```

Exercise 7.17: vowelCount

Write a static method named vowelCount that accepts a String as a parameter and returns an array of integers representing the counts of each vowel in the String. The array returned by your method should hold 5 elements: the first is the count of As, the second is the count of Es, the third Is, the fourth Os, and the fifth Us. Assume that the string contains no uppercase letters.

For example, the call vowelCount("i think, therefore i am") should return the array {1, 3, 3, 1, 0}.

```
public static int[] vowelCount(String text) {
    int[] counts = new int[5];
    for (int i = 0; i < text.length(); i++) {
        char c = text.charAt(i);
        if (c == 'a') {
            counts[0]++;
        } else if (c == 'e') {
            counts[1]++;
        } else if (c == 'i') {
            counts[2]++;
        } else if (c == 'o') {
            counts[3]++;
        } else if (c == 'u') {
            counts[4]++;
        }
    }
    return counts;
}
```

Exercise 7.18: wordLengths

Write a method called wordLengths that accepts a Scanner representing an input file as its argument. Your method should read from the given file, count the number of letters in each token in the file, and output a result diagram of how many words contain each number of letters. **Use tabs** before the asterisks so that they'll line up. If there are no words of a given length, omit that line from the output.

For example, if the file contains the following text:

```
Before sorting:
13 23 480 -18 75
```

```
hello how are you feeling today

After sorting:
-18 13 23 75 480
are feeling hello how today you
```

your method should produce the following output to the console:

```
2: 6    ******
3: 10   **********
5: 5    *****
6: 1    *
7: 2    **
8: 2    **
```

You may assume that no token in the file is more than 80 characters in length.

```
public static void wordLengths(Scanner console) {
   int[] array = new int[81];

   while (console.hasNext()) {
      array[console.next().length()]++;
   }

   for (int i = 1; i < array.length; i++) {
      if (array[i] != 0) {
         System.out.printf("%d: %d\t", i, array[i]);
         for (int j = 0; j < array[i]; j++) {
            System.out.print("*");
         }
         System.out.println();
      }
   }
}
```

Exercise 7.19: matrixAdd
Write a method named matrixAdd that accepts a pair of two-dimensional arrays of integers as parameters, treats the arrays as 2D matrices and adds them, returning the result. The sum of two matrices A and B is a matrix C where for every row i and column j, $C_{ij} = A_{ij} + B_{ij}$. You may assume that the arrays passed as parameters have the same dimensions.

```
public static int[][] matrixAdd(int[][] a, int[][] b) {
   int rows = a.length;
   int cols = 0;
```

```
    if (rows > 0) {
        cols = a[0].length;
    }
    int[][] c = new int[rows][cols];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            c[i][j] = a[i][j] + b[i][j];
        }
    }
    return c;
}
```

Exercise 7.20: isMagicSquare
Write a method called isMagicSquare that accepts a two-dimensional array of integers as a parameter and returns true if it is a magic square. A square matrix is a magic square if it is square in shape (same number of rows as columns, and every row the same length), and all of its row, column, and diagonal sums are equal. For example, [[2, 7, 6], [9, 5, 1], [4, 3, 8]] is a magic square because all eight of the sums are exactly 15.

```
public static boolean isMagicSquare(int[][] a) {
    if (a.length == 0 || a.length == 1 && a[0].length == 1) {
        return true;   // trivial cases; empty and one-element array
    }

    // get sum of first row, use as basis for others
    int sum = 0;
    for (int c = 0; c < a[0].length; c++) {
        sum += a[0][c];
    }

    // compute all row sums
    for (int r = 0; r < a.length; r++) {
        if (a[r].length != a.length) {
            return false;   // not square shape
        }
        int rowSum = 0;
        for (int c = 0; c < a[r].length; c++) {
            rowSum += a[r][c];
        }
        if (rowSum != sum) {
            return false;
        }
    }

    // compute all column sums
    for (int c = 0; c < a.length; c++) {
        int colSum = 0;
        for (int r = 0; r < a.length; r++) {
            colSum += a[r][c];
        }
        if (colSum != sum) {
            return false;
        }
    }

    // compute diagonal sums
    int diagSum1 = 0;
    int diagSum2 = 0;
    for (int rc = 0; rc < a.length; rc++) {
        diagSum1 += a[rc][rc];
        diagSum2 += a[rc][a.length - 1 - rc];
    }
    if (diagSum1 != sum || diagSum2 != sum) {
        return false;
    }

    return true;
}
```

Chapter 8: Classes

Exercise 8.1: quadrantPoint

dd the following method to the Point class:

```
public int quadrant()
```

Returns which quadrant of the x/y plane this Point object falls in. Quadrant 1 contains all points whose x and y values are both positive. Quadrant 2 contains all points with negative x but positive y. Quadrant 3 contains all points with negative x and y values. Quadrant 4 contains all points with positive x but negative y. If the point lies directly on the x and/or y axis, return 0.

```
public class Point {
   private int x;
   private int y;

   // // your code goes here

}
```

```
public int quadrant() {
   if (x > 0 && y > 0) {
      return 1;
   } else if (x < 0 && y > 0) {
      return 2;
   } else if (x < 0 && y < 0) {
      return 3;
   } else if (x > 0 && y < 0) {
      return 4;
   } else {
      return 0;
   }
}
```

Exercise 8.2: flipPoint

Add the following method to the Point class:

```
public void flip()
```

Negates and swaps the x/y coordinates of the Point object. For example, if the object initially represents the point (5, -3), after a call to flip, the object should represent (3, -5). If the object initially represents the point (4, 17), after a call to flip, the object should represent (-17, -4).

```
public class Point {
    private int x;
    private int y;

    // // your code goes here

}

public void flip() {
    int temp = x;
    x = -y;
    y = -temp;
}
```

Exercise 8.3: manhattanDistancePoint

Add the following method to the Point class:

```
public int manhattanDistance(Point other)
```

Returns the "Manhattan distance" between the current Point object and the given other Point object. The Manhattan distance refers to how far apart two places are if the person can only travel straight horizontally or vertically, as though driving on the streets of Manhattan. In our case, the Manhattan distance is the sum of the absolute values of the differences in their coordinates; in other words, the difference in x plus the difference in y between the points.

```
public class Point {
    private int x;
    private int y;

    // // your code goes here

}

// Returns the "Manhattan (rectangular) distance" between
// this point and the given other point.
public int manhattanDistance(Point other) {
    int dx = x - other.x;
    int dy = y - other.y;
    return Math.abs(dx) + Math.abs(dy);
}
```

Exercise 8.4: isVerticalPoint

Add the following method to your Point class:

public boolean isVertical(Point other)

Returns true if the given Point lines up vertically with this Point; that is, if their x coordinates are the same.

```
// Returns true if the given point lines up vertically
// with this point (if they have the same x value).
public boolean isVertical(Point p) {
   return x == p.x;
}
```

Exercise 8.5: slopePoint

Add the following method to the Point class:

public double slope(Point other)

Returns the slope of the line drawn between this Point and the given other Point. Use the formula (*y*2 - *y*1) / (*x*2 - *x*1) to determine the slope between two points (*x*1, *y*1) and (*x*2, *y*2). Note that this formula fails for points with identical x-coordinates, so throw an IllegalArgumentException in this case.

```
public class Point {
   private int x;
   private int y;

   // // your code goes here

}
```

```
public double slope(Point other) {
   if (x == other.x) {
      throw new IllegalArgumentException();
   }
   return (other.y - y) / (1.0 * other.x - x);
}
```

Exercise 8.6: isCollinearPoint

Add the following method to the Point class:

public boolean **isCollinear**(Point p1, Point p2)

Returns whether this Point is collinear with the given two other points. Points are collinear if a straight line can be drawn that connects them. Two basic examples are three points that have the same x- or y-coordinate. The more general case can be determined by calculating the slope of the line between each pair of points and checking whether this slope is the same for all pairs of points. Use the formula (*y*2 - *y*1) / (*x*2 - *x*1) to determine the slope between two points (*x*1, *y*1) and (*x*2, *y*2). (Note that this formula fails for points with identical x-coordinates so this will have to be special-cased in your code.)

Since Java's double type is imprecise, round all slope values to a reasonable accuracy such as four digits past the decimal point before you compare them.

```
public class Point {
   private int x;
   private int y;

   // // your code goes here

}

public boolean isCollinear(Point p1, Point p2) {
   // basic case: all points have same x or y value
   if ((x == p1.x && x == p2.x) || (y == p1.y && y == p2.y)) {
      return true;
   }
   // complex case: compare slopes
   double slope1 = (p1.y - y) / (p1.x - x);
   double slope2 = (p2.y - y) / (p2.x - x);
   return round(slope1, 4) == round(slope2, 4);
}
private double round(double value, int places) {
   for (int i = 0; i < places; i++) {
      value *= 10.0;
   }
   value = Math.round(value);
   for (int i = 0; i < places; i++) {
      value /= 10.0;
   }
   return value;
}
```

Exercise 8.7: addTimeSpan

Add the following method to the TimeSpan class:

```
public void add(TimeSpan span)
```

Adds the given amount of time to this time span.

Recall the TimeSpan code from this chapter:

```
public class TimeSpan {
    private int hours;
    private int minutes;

    public void add(int hours, int minutes) { ... }
    public int getHours() { ... }
    public int getMinutes() { ... }
    public String toString() { ... }

    // // your code goes here

}
```

```
// Adds the amount of time represented by the given time
// span to this time span.
public void add(TimeSpan time) {
    add(time.hours, time.minutes);
}
```

Exercise 8.8: subtractTimeSpan

Add the following method to the TimeSpan class:

```
public void subtract(TimeSpan span)
```

Subtracts the given amount of time from this time span. (You do not need to worry about the case of a negative result.)

Recall the TimeSpan code from this chapter:

```
public class TimeSpan {
    private int hours;
    private int minutes;

    public void add(int hours, int minutes) { ... }
    public int getHours() { ... }
    public int getMinutes() { ... }
    public String toString() { ... }

    // // your code goes here

}
```

```
public void subtract(TimeSpan span) {
    hours -= span.hours;
    minutes -= span.minutes;
    if (minutes < 0) {
        minutes = minutes + 60;
        hours--;
    }
}
```

Exercise 8.9: scaleTimeSpan

Add the following method to the TimeSpan class:

```
public void scale(int factor)
```

Scales this time span by the given factor. For example, 1 hour and 45 minutes scaled by 2 equals 3 hours and 30 minutes.

Recall the TimeSpan code from this chapter:

```
public class TimeSpan {
    private int hours;
    private int minutes;

    public void add(int hours, int minutes) { ... }
    public int getHours() { ... }
    public int getMinutes() { ... }
    public String toString() { ... }

    // // your code goes here

}

// Adds the given interval to this time span.
// pre: hours >= 0 and minutes >= 0
public void scale(int factor) {
    hours *= factor;
    minutes *= factor;
    // convert any overflow of 60 minutes into one hour
    hours += minutes / 60;
    minutes = minutes % 60;
}
```

Exercise 8.10: clearStock

Add the following mutator method to the Stock class from textbook section 8.5:

```
public void clear()
```

Resets this Stock's number of shares purchased and total cost to 0.

```
// Removes all purchases of this Stock.
public void clear() {
   totalShares = 0;
   totalCost = 0.00;
}
```

Exercise 8.11: transcationFeeBankAccount

Suppose that you are provided with a pre-written class BankAccount as shown below. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

Write an instance method named transactionFee that will be placed inside the BankAccount class to become a part of each BankAccount object's behavior. The transactionFee method accepts a fee amount (a real number) as a parameter, and applies that fee to the user's past transactions. The fee is applied once for the first transaction, twice for the second transaction, three times for the third, and so on. These fees are subtracted out from the user's overall balance. If the user's balance is large enough to afford all of the fees with greater than $0.00 remaining, the method returns true. If the balance cannot afford all of the fees or has no money left, the balance is left as 0.0 and the method returns false.

```
// A BankAccount keeps track of a user's money balance and ID,
// and counts how many transactions (deposits/withdrawals) are made.
public class BankAccount {
   private String id;
   private double balance;
   private int transactions;

   // Constructs a BankAccount object with the given id, and
   // 0 balance and transactions.
   public BankAccount(String id)

   // returns the field values
   public double getBalance()
   public String getID()
   public String getTransactions()

   // Adds the amount to the balance if it is between 0-500.
   // Also counts as 1 transaction.
   public void deposit(double amount)
```

```
    // Subtracts the amount from the balance if the user has enough money.
    // Also counts as 1 transaction.
    public void withdraw(double amount)

    // your method would go here
}
```

For example, given the following BankAccount object:

```
BankAccount savings = new BankAccount("Jimmy");
savings.deposit(10.00);
savings.deposit(50.00);
savings.deposit(10.00);
savings.deposit(70.00);
```

The account at that point has a balance of $140.00. If the following call were made:

```
savings.transactionFee(5.00);
```

Then the account would be deducted $5 + $10 + $15 + $20 for the four transactions, leaving a final balance of $90.00. The method would return true. If a second call were made,

```
savings.transactionFee(10.00);
```

Then the account would be deducted $10 + $20 + $30 + $40 for the four transactions, leaving a final balance of $0.00. The method would return false.

```
public boolean transactionFee(double amount) {
    for (int i = 1; i <= transactions; i++) {
        balance -= amount * i;
    }
    if (balance > 0.0) {
        return true;
    } else {
        balance = 0.0;
        return false;
    }
}
```

Exercise 8.12: toStringBankAccount

Add the following method to the BankAccount class:

```
public String toString()
```

Your method should return a string that contains the account's name and balance separated by a comma and space. For example, if an account object named benben has the name "Benson" and a balance of 17.25, the call of benben.toString() should return:

```
Benson, $17.25
```

There are some special cases you should handle. If the balance is negative, put the - sign before the dollar sign. Also, always display the cents as a two-digit number. For example, if the same object had a balance of -17.5, your method should return:

```
Benson, -$17.50
```

Your code is being added to the following class:

```java
public class BankAccount {
    private String name;
    private double balance;

    // // your code goes here

}

public String toString() {
    String result = name + ", ";

    // handle negative numbers
    if (balance < 0) {
        result += "-";
    }

    result += "$" + Math.abs(balance);

    // handle numbers with cents a multiple of ten (e.g. $12.30)
    if (result.charAt(result.length() - 2) == '.') {
        result += "0";
    }

    return result;
}
```

Exercise 8.13: transferBankAccount

Suppose that you are provided with a pre-written class BankAccount as shown below. (The headings are shown, but not the method bodies, to save space.) Assume that the fields, constructor, and methods shown are already implemented. You may refer to them or use them in solving this problem if necessary.

Write an instance method named transfer that will be placed inside the BankAccount class to become a part of each BankAccount object's behavior. The transfer method moves money from this bank account to another account. The method accepts two parameters: a second BankAccount to accept the money, and a real number for the amount of money to transfer.

There is a $5.00 fee for transferring money, so this much must be deducted from the current account's balance before any transfer.

The method should modify the two BankAccount objects such that "this" current object has its balance decreased by the given amount plus the $5 fee, and the other BankAccount object's balance is increased by the given amount. A transfer also counts as a transaction on both accounts.

If this account object does not have enough money to make the full transfer, transfer whatever money is left after the $5 fee is deducted. If this account has under $5 or the amount is 0 or less, no transfer should occur and neither account's state should be modified.

```
// A BankAccount keeps track of a user's money balance and ID,
// and counts how many transactions (deposits/withdrawals) are made.
public class BankAccount {
   private String id;
   private double balance;
   private int transactions;

   // Constructs a BankAccount object with the given id, and
   // 0 balance and transactions.
   public BankAccount(String id)

   // returns the field values
   public double getBalance()
   public String getID()
   public String getTransactions()

   // Adds the amount to the balance if it is between 0-500.
   // Also counts as 1 transaction.
```

```
    public void deposit(double amount)

    // Subtracts the amount from the balance if the user has enough money.
    // Also counts as 1 transaction.
    public void withdraw(double amount)

    // your method would go here
}
```

For example, given the following BankAccount objects:

```
BankAccount ben = new BankAccount("Benson");
ben.deposit(90.00);
BankAccount mar = new BankAccount("Marty");
mar.deposit(25.00);
```

Assuming that the following calls were made, the balances afterward are shown in comments to the right of each call:

```
ben.transfer(mar, 20.00);   // ben $65, mar $45   (ben loses $25, mar gains $20)
ben.transfer(mar, 10.00);   // ben $50, mar $55   (ben loses $15, mar gains $10)
ben.transfer(mar, -1);      // ben $50, mar $55   (no effect; negative amount)
mar.transfer(ben, 39.00);   // ben $89, mar $11   (mar loses $44, ben gains $39)
mar.transfer(ben, 50.00);   // ben $95, mar $ 0   (mar loses $11, ben gains $ 6)
mar.transfer(ben,  1.00);   // ben $95, mar $ 0   (no effect; no money in account)
ben.transfer(mar, 88.00);   // ben $ 2, mar $88   (ben loses $93, mar gains $88)
ben.transfer(mar,  1.00);   // ben $ 2, mar $88   (no effect; can't afford fee)
```

```
public void transfer(BankAccount other, double amount) {
    if (amount > 0.00) {
        if (amount + 5.00 <= balance) {
            withdraw(amount + 5.00);
            other.deposit(amount);
        } else {
            if (amount > 5.00) {
                // partial transfer
                other.deposit(balance - 5.00);
                withdraw(balance);
            }
        }
    }
}
```

Exercise 8.14: calssLine

Write a class called Line that represents a line segment between two Points. Your Line objects should have the following methods:

public Line(Point p1, Point p2)

Constructs a new line that contains the given two points.

public Point **getP1**()

Returns this line's first endpoint.

public Point **getP2**()

Returns this line's second endpoint.

public String **toString**()

Returns a string representation of this line, such as "[(22, 3), (4, 7)]".

```
public class Line {
   private Point p1;
   private Point p2;

   public Line(Point p1, Point p2) {
      this.p1 = p1;
      this.p2 = p2;
   }

   public Point getP1() {
      return p1;
   }

   public Point getP2() {
      return p2;
   }

   public String toString() {
      return "[(" + p1.x + ", " + p1.y +"), (" + p2.x + ", " + p2.y + ")]";
   }
}
```

Exercise 8.15: getSlopLine

Add the following method to your Line class from the previous exercise:

public double **getSlope**()

Returns the slope of this line. The slope of a line between points (x_1, y_1) and (x_2, y_2) is equal to (y_2 - y_1) / (x_2 - x_1). If the two points have the same *x*-coordinates, the denominator is zero and the slope is undefined, so you should throw an IllegalStateException in this case.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write the method's complete code in the box provided.) See the previous exercise for a description of the Line class and its public members.
Type your solution here:

```
public double getSlope() {
   double x1 = p1.getX();
   double x2 = p2.getX();
   double y1 = p1.getY();
   double y2 = p2.getY();
   if (x1 == x2) {
      throw new IllegalStateException();
   } else if (y1 == y2) {
      return 0;
   } else {
      return (y2 - y1) / (x2 - x1);
   }
}
```

Exercise 8.16: constructorLine
Add the following constructor to your Line class from the preceding exercises:

public Line(int x1, int y1, int x2, int y2)

Constructs a new line that contains the given two points.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write your constructor's complete code in the box provided.) See the previous exercises for a description of the Line class and its public members.

```
// Constructs a new Line that contains the given two points.
public Line(int x1, int y1, int x2, int y2) {
   p1 = new Point(x1, y1);
   p2 = new Point(x2, y2);
}
```

Exercise 8.17: isCollinearLine

Add the following method to your Line class:

public boolean **isCollinear**(Point p)

Returns true if the given point is collinear with the points of this line; that is, if, when this Line is stretched infinitely, it would eventually hit the given point. Points are collinear if a straight line can be drawn that connects them. Two basic examples are three points that have the same x- or y-coordinate. The more general case can be determined by calculating the slope of the line between each pair of points and checking whether this slope is the same for all pairs of points. Use the formula (y2 - y1) / (x2 - x1) to determine the slope between two points (x1, y1) and (x2, y2).

(Note that this formula fails for points with identical x-coordinates, so this will have to be a special case in your code.) Since Java's double type is imprecise, round all slope values to four digits past the decimal point before you compare them.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write your constructor's complete code in the box provided.) See the previous exercises for a description of the Line class and its public members.

```
public boolean isCollinear(Point p) {
   double slope1 = (double)(p1.getY() - p.getY()) / (p1.getX() - p.getX());
   double slope2 = (double)(p2.getY() - p.getY()) / (p2.getX() - p.getX());
   return slope1 == slope2;
}
```

Exercise 8.18: classRectangle
Write a class called Rectangle that represents a rectangular two-dimensional region.
Your Rectangle objects should have the following methods:

public Rectangle(int x, int y, int width, int height)

Constructs a new rectangle whose top-left corner is specified by the given coordinates and with the given width and height. Throw an IllegalArgumentException on a negative width or height.

public int getHeight()

Returns this rectangle's height.

public int getWidth()

Returns this rectangle's width.

public int getX()

Returns this rectangle's x-coordinate.

public int getY()

Returns this rectangle's y-coordinate.

public String toString()

Returns a string representation of this rectangle, such as "Rectangle[x=1,y=2,width=3,height=4]".

```
// Represents a 2-dimensional rectangular region.
public class Rectangle {
   private int x;
   private int y;
   private int width;
   private int height;
   // Constructs a new Rectangle whose top-left corner is specified by the
   // given coordinates and with the given width and height.
   public Rectangle(int x, int y, int width, int height) {
      if (width < 0 || height < 0) {
         throw new IllegalArgumentException();
      }
      this.x = x;
      this.y = y;
      this.width = width;
      this.height = height;
   }
   // Returns this Rectangle's height.
   public int getHeight() {
      return height;
   }
   // Returns this Rectangle's width.
   public int getWidth() {
      return width;
   }
   // Returns this Rectangle's x coordinate.
   public int getX() {
      return x;
   }
   // Returns this Rectangle's y coordinate.
```

```
    public int getY() {
        return y;
    }
    // Returns a String representation of this Rectangle, such as
    // "Rectangle[x=1,y=2,width=3,height=4]".
    public String toString() {
        return "Rectangle[x=" + x + ",y=" + y +
               ",width=" + width + ",height=" + height + "]";
    }
}
```

Exercise 8.19: constructorRectangle

Add the following constructor to your Rectangle class from the previous exercise:

```
public Rectangle(Point p, int width, int height)
```

Constructs a new rectangle whose top-left corner is specified by the given point and with the given width and height.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write your constructor's complete code in the box provided.) See previous exercises for a description of the Rectangle and Point classes and their public members.

```
// Constructs a new rectangle whose top-left corner is specified by the
// given point and with the given width and height.
public Rectangle(Point p, int width, int height) {
    this.x = p.x;
    this.y = p.y;
    this.width = width;
    this.height = height;
}
```

Exercise 8.20: containsRectangle

Add the following accessor methods to your Rectangle class from the previous exercises:

```
public boolean contains(int x, int y)
public boolean contains(Point p)
```

Returns whether the given Point or coordinates lie inside the bounds of this Rectangle. The edges are included; for example, a rectangle with [x=2,y=5,width=8,height=10] will return true for any point from (2, 5) through (10, 15) inclusive.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write your methods' complete code in the box provided.) See previous exercises for a description of the Rectangle and Point classes and their public members.

```
public boolean contains(int x, int y) {
   return (x <= this.x + width && x >= this.x) && (y <= this.y + height && y >= this.y);
}

public boolean contains(Point p) {
   return (p.getX() <= this.x + width && p.getX() >= this.x) && (p.getY() <= this.y + height &
& p.getY() >= this.y);
}
```

Exercise 8.21: unionRectangle
Add the following method to your Rectangle class from the previous exercises:

public Rectangle **union**(Rectangle rect)

Returns a new Rectangle that represents the area occupied by the tightest bounding box that contains both this Rectangle and the given other Rectangle. Your method should not modify the current Rectangle or the one that is passed in as a parameter; you should create and return a new rectangle.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write your method's complete code in the box provided.) See previous exercises for a description of the Rectangle class and its public members.

```
// Returns a new Rectangle that represents the tightest bounding box
// that contains both this rectangle and the other rectangle.
public Rectangle union(Rectangle rect) {
   int left = Math.min(x, rect.x);
   int top = Math.min(y, rect.y);
   int right = Math.max(x + width, rect.x + rect.width);
   int bottom = Math.max(y + height, rect.y + rect.height);
   return new Rectangle(left, top, right - left, bottom - top);
}
```

Exercise 8.22: intersectionRectangle
Add the following method to your Rectangle class from the previous exercises:

public Rectangle **intersection**(Rectangle rect)

Returns a new rectangle that represents the largest rectangular region completely contained within both this rectangle and the given other rectangle. If the rectangles do not intersect at all, returns null. Your method should not modify the current Rectangle or the one that is passed in as a parameter; you should create and return a new rectangle.

(You don't need to write the class header or declare the fields; assume that this is already done for you. Just write your method's complete code in the box provided.) See the previous exercises for a description of the Rectangle class and its public members.

```
// Returns a new rectangle that represents the largest rectangular region
// completely contained within both this rectangle and the given other
// rectangle.  If the rectangles do not intersect at all, returns a rectangle
// whose width and height are both 0.
public Rectangle intersection(Rectangle rect) {
   int left = Math.max(x, rect.x);
   int top = Math.max(y, rect.y);
   int right = Math.min(x + width, rect.x + rect.width);
   int bottom = Math.min(y + height, rect.y + rect.height);
   int width = Math.max(0, right - left);
   int height = Math.max(0, bottom - top);
   return new Rectangle(left, top, width, height);
}
```

Chapter 9: Inheritance and Interfaces

Exercise 9.1: Marketer

Write the class Marketer to accompany the other law firm classes described in this chapter. Marketers make $50,000 ($10,000 more than general employees) and have an additional method called advertise that prints "Act now, while supplies last!" Make sure to interact with the Employee superclass as appropriate.

```
// A class to represent marketers.
public class Marketer extends Employee {
   public void advertise() {
      System.out.println("Act now, while supplies last!");
   }
   public double getSalary() {
      return super.getSalary() + 10000;
   }
}
```

Exercise 9.2: Janitor

Write a class Janitor to accompany the other law firm classes described in this chapter. Janitors work twice as many hours per week as other employees (80 hours/week), they make $30,000 ($10,000 less than general employees), they get half as much vacation as other employees (only 5 days), and they have an additional method clean that prints "Workin' for the man." Make sure to interact with the superclass as appropriate.

```
// A class to represent marketers.
public class Janitor extends Employee {
   public void clean() {
      System.out.println("Workin' for the man.");
   }
   public int getHours() {
      return super.getHours() * 2;
   }
   public double getSalary() {
      return super.getSalary() - 10000.00;
   }
   public int getVacationDays() {
      return super.getVacationDays() / 2;
   }
}
```

Exercise 9.3: HarvardLawyer

Write a class HarvardLawyer to accompany the other law firm classes described in this chapter. Harvard lawyers are like normal lawyers, but they make 20% more money than a normal lawyer, they get 3 days more vacation, and they have to fill out four of the lawyer's forms to go on vacation. That is, the getVacationForm method should return "pinkpinkpinkpink". Make sure to interact with the superclass as appropriate.

```
// A class to represent Harvard lawyers.
public class HarvardLawyer extends Lawyer {
    public double getSalary() {
        return super.getSalary() * 1.2;
    }
    public int getVacationDays() {
        return super.getVacationDays() + 3;
    }
    public String getVacationForm() {
        String lawyerForm = super.getVacationForm();
        return lawyerForm + lawyerForm + lawyerForm + lawyerForm;
    }
}
```

Exercise 9.4: MonsterTruck

Suppose that the following two classes have been declared:

```
public class Car {
    public void m1() {
        System.out.println("car 1");
    }

    public void m2() {
        System.out.println("car 2");
    }

    public String toString() {
        return "vroom";
    }
}
public class Truck extends Car {
    public void m1() {
        System.out.println("truck 1");
    }

    public void m2() {
        super.m1();
    }

    public String toString() {
        return super.toString() + super.toString();
    }
}
```

Write a class MonsterTruck whose methods have the behavior below. Don't just print/return the output; whenever possible, use inheritance to reuse behavior from the superclass.

Method	Output/Return
m1	monster 1
m2	truck 1 car 1
toString	"monster vroomvroom"

```
public class MonsterTruck extends Truck {
   public void m1() {
      System.out.println("monster 1");
   }

   public void m2() {
      super.m1();
      super.m2();
   }

   public String toString() {
      return "monster " + super.toString();
   }
}
```

Exercise 9.9: MinMaxAccount

A company has written a large class BankingAccount with many methods including:

Method/Constructor	Description
public BankingAccount(Startup *s*)	constructs a BankingAccount object using information in the Startup object *s*
public void debit(Debit *d*)	records the given debit
public void credit(Credit *c*)	records the given credit
public int getBalance()	returns current balance in pennies

Design a new class MinMaxAccount whose instances can be used in place of a BankingAccount object but include new behavior of remembering the minimum and maximum balances ever recorded for the account. You should provide the same methods as the superclass, as well as the following new behavior:

Method/Constructor	Description

public MinMaxAccount(Startup *s*)	constructs a MinMaxAccount object using information in the Startup object *s*
public int getMin()	returns minimum balance in pennies
public int getMax()	returns maximum balance in pennies

The account's constructor sets the initial balance based on the Startup information. Assume that only the debit and credit methods change an account's balance.

```
public class MinMaxAccount extends BankingAccount {
   private int minBalance;
   private int maxBalance;
   public MinMaxAccount(Startup s) {
      super(s);
      minBalance = getBalance();
      maxBalance = getBalance();
   }
   public void debit(Debit d) {
      super.debit(d);
      updateMinMax();
   }
   public void credit(Credit c) {
      super.credit(c);
      updateMinMax();
   }
   private void updateMinMax() {
      int balance = getBalance();
      if (balance < minBalance) {
         minBalance = balance;
      } else if(balance > maxBalance) {
         maxBalance = balance;
      }
   }
   public int getMin() {
      return minBalance;
   }
   public int getMax() {
      return maxBalance;
   }
}
```

Exercise 9.10: DiscountBill

Suppose a class GroceryBill keeps track of a list of items being purchased at a market:

Method/Constructor	Description
public GroceryBill(Employee *clerk*)	constructs a GroceryBill object for the given *clerk*
public void add(Item *i*)	adds *i* to this bill's total
public double getTotal()	returns the cost of these items
public void printReceipt()	prints a list of items

GroceryBill objects interact with Item objects. An Item has the following public methods:

Method/Constructor	Description
public double getPrice()	returns the price for this item
public double getDiscount()	returns the discount for this item

For example, a candy bar item might cost 1.35 with a discount of 0.25 for preferred customers, meaning that preferred customers get it for 1.10. (Some items will have no discount, 0.0.) Currently the above classes do not consider discounts. Every item in a bill is charged full price, and item discounts are ignored.

Define a class DiscountBill that extends GroceryBill to compute discounts for preferred customers. The constructor for DiscountBill accepts a parameter for whether the customer should get the discount.

Your class should adjust the amount reported by getTotal for preferred customers. For example, if the total would have been $80 but a preferred customer is getting $20 in discounts, then getTotal should report the total as $60 for that customer. You should also keep track of how many items a customer is getting a non-zero discount for and the overall discount, both as a total amount and as a percentage of the original bill. Include the extra methods below that allow a client to ask about the discount:

Method/Constructor	Description
public DiscountBill(Employee *clerk*, boolean *preferred*)	constructs discount bill for given *clerk*
public int getDiscountCount()	returns the number of items that were discounted, if any
public double getDiscountAmount()	returns the total discount for this list of items, if any

public double getDiscountPercent()	returns the percent of the total discount as a percent of what the total would have been otherwise

If the customer is not a preferred customer the DiscountBill behaves at all times as if there is a total discount of 0.0 and no items have been discounted.

```
public class DiscountBill extends GroceryBill {
   private boolean preferred;
   private int count;
   private double discount;
   public DiscountBill(Employee clerk, boolean preferred) {
      super(clerk);
      this.preferred = preferred;
      count = 0;
      discount = 0.0;
   }
   public void add(Item i) {
      super.add(i);
      if (preferred) {
         double amount = i.getDiscount();
         if (amount > 0.0) {
            count++;
            discount += amount;
         }
      }
   }
   public double getTotal() {
      return super.getTotal() - discount;
   }
   public int getDiscountCount() {
      return count;
   }
   public double getDiscountAmount() {
      return discount;
   }
   public double getDiscountPercent() {
      return discount / super.getTotal() * 100;
   }
}
```

Exercise 9.11: FilteredAccount

A cash processing company has a class called Account used to process transactions:

Method/Constructor	Description
public Account(Client c)	constructs an account using client information
public boolean process(Transaction t)	processes the next transaction, returning true if transaction was approved, false otherwise

Account objects interact with Transaction objects, which have many methods including:

Method/Constructor	Description
public int value()	returns the value of this transaction in pennies (could be negative, positive or zero)

The company wishes to create a slight modification to the Account class that filters out zero-valued transactions. Design a new class called FilteredAccount whose instances can be used in place of an Account object but which include the extra behavior of not processing transactions with a value of 0. More specifically, the new class should indicate that a zero-valued transaction was approved but shouldn't call the process method in the Account class to process it. Your class should have a single constructor that accepts a parameter of type Client, and it should include the following method:

Method/Constructor	Description
public double percentFiltered()	returns the percent of transactions filtered out (between 0.0 and 100.0); returns 0.0 if no transactions submitted

Assume that all transactions enter the system by a call on the process method described above.

```
public class FilteredAccount extends Account {
   private int zeros;
   private int transactions;
   public FilteredAccount(Client c) {
      super(c);
      zeros = 0;
      transactions = 0;
   }
   public boolean process(Transaction t) {
      transactions++;
      if (t.value() == 0) {
         zeros++;
         return true;
      } else {
         return super.process(t);
```

```
        }
    }
    public double percentFiltered() {
        if (transactions == 0) {
            return 0.0;
        } else {
            return zeros * 100.0 / transactions;
        }
    }
}
```

Chapter 10: ArrayLists

Exercise 10.2: swapPairs

Write a method swapPairs that switches the order of values in an ArrayList of Strings in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. For example, if the list initially stores these values: {"four", "score", "and", "seven", "years", "ago"} your method should switch the first pair, "four", "score", the second pair, "and", "seven", and the third pair, "years", "ago", to yield this list: {"score", "four", "seven", "and", "ago", "years"}

If there are an odd number of values in the list, the final element is not moved. For example, if the original list had been: {"to", "be", "or", "not", "to", "be", "hamlet"} It would again switch pairs of values, but the final value, "hamlet" would not be moved, yielding this list: {"be", "to", "not", "or", "be", "to", "hamlet"}

```
public void swapPairs(ArrayList<String> list) {
   for (int i = 0; i < list.size() - 1; i += 2) {
      String first = list.remove(i);
      list.add(i + 1, first);
   }
}
```

Exercise 10.3: removeEvenLength

Write a method removeEvenLength that takes an ArrayList of Strings as a parameter and that removes all of the strings of even length from the list.

```
public void removeEvenLength(ArrayList<String> list) {
   int i = 0;
   while (i < list.size()) {
      if (list.get(i).length() % 2 == 0) {
         list.remove(i);
      } else {
         i++;
      }
   }
}
```

Exercise 10.4: doubleList

Write a method doubleList that takes an ArrayList of Strings as a parameter and that replaces every string with two of that string. For example, if the list stores the values {"how", "are", "you?"} before the method is called, it should store the values {"how", "how", "are", "are", "you?", "you?"} after the method finishes executing.

```
public void doubleList(ArrayList<String> list) {
    for (int i = 0; i < list.size(); i += 2) {
        list.add(i, list.get(i));
    }
}
```

Exercise 10.6: minToFront

Write a method minToFront that takes an ArrayList of integers as a parameter and that moves the minimum value in the list to the front, otherwise preserving the order of the elements. For example, if a variable called list stores the following values: {3, 8, 92, 4, 2, 17, 9} and you make this call: minToFront(list); it should store the following values after the call: {2, 3, 8, 92, 4, 17, 9} You may assume that the list stores at least one value.

```
public void minToFront(ArrayList<Integer> list) {
    int min = 0;
    for (int i = 1; i < list.size(); i++){
        if (list.get(i) < list.get(min)) {
            min = i;
        }
    }
    list.add(0, list.remove(min));
}
```

Exercise 10.7: removeDuplicates

Write a method removeDuplicates that takes as a parameter a sorted ArrayList of Strings and that eliminates any duplicates from the list. For example, suppose that a variable called list contains the following values: {"be", "be", "is", "not", "or", "question", "that", "the", "to", "to"} After calling removeDuplicates(list); the list should store the following values: {"be", "is", "not", "or", "question", "that", "the", "to"}

Because the values will be sorted, all of the duplicates will be grouped together.

```
public void removeDuplicates(ArrayList<String> list) {
    int index = 0;
    while (index < list.size() - 1) {
        String s1 = list.get(index);
        String s2 = list.get(index + 1);
        if (s1.equals(s2)) {
            list.remove(index + 1);
        } else {
            index++;
        }
    }
}
```

Exercise 10.10: removeInRange

Write a method called removeInRange that accepts four parameters: an ArrayList of integers, an element value, a starting index, and an ending index. The method's behavior is to remove all occurrences of the given element that appear in the list between the starting index (inclusive) and the ending index (exclusive). Other values and occurrences of the given value that appear outside the given index range are not affected.

For example, for the list [0, 0, 2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0, 16], a call of removeInRange(list, 0, 5, 13); should produce the list [0, 0, 2, 0, 4, 6, 8, 10, 12, 0, 14, 0, 16]. Notice that the zeros located at indices between 5 inclusive and 13 exclusive in the original list (before any modifications were made) have been removed.

```
public static void removeInRange(ArrayList<Integer> array, int value, int start, int end) {

   for (int i = start; i < end; i++) {
      if (array.get(i) == value) {
         array.remove(i);
         i--;
         end--;
      }
   }

}
```

Exercise 10.11: stutter

Write a method stutter that takes an ArrayList of Strings and an integer *k* as parameters and that replaces every string with *k* copies of that string. For example, if the list stores the values ["how", "are", "you?"] before the method is called and *k* is 4, it should store the values ["how", "how", "how", "how", "are", "are", "are", "are", "you?", "you?", "you?", "you?"] after the method finishes executing. If *k* is 0 or negative, the list should be empty after the call.

```
public static void stutter(ArrayList<String> array, int k) {
   if (k <= 0) {
      array.clear();
   } else {
      int oldSize = array.size();

      for (int i = 0; i < oldSize; i++) {
         String word = array.get(i * k);
         for (int j = 1; j < k; j++) {
            array.add(i * k + j, word);
         }
      }
```

```
    }
}
```

Exercise 10.12 markLength4

Write a method markLength4 that takes an ArrayList of Strings as a parameter and that places a string of four asterisks "****" in front of every string of length 4. For example, suppose that a variable called list contains the following values: {"this", "is", "lots", "of", "fun", "for", "every", "Java", "programmer"} And you make the following call: markLength4(list); then list should store the following values after the call: {"****", "this", "is", "****", "lots", "of", "fun", "for", "every", "****", "Java", "programmer"}

Notice that you leave the original strings in the list, "this", "lots", "Java", but include the four-asterisk string in front of each to mark it.

```
public void markLength4(ArrayList<String> list) {
    int index = 0;
    while (index < list.size()) {
        if (list.get(index).length() == 4) {
            list.add(index, "****");
            index += 2;
        } else {
            index++;
        }
    }
}
```

Exercise 10.14: removeShorterStrings

Write a method removeShorterStrings that takes an ArrayList of Strings as a parameter and that removes from each successive pair of values the shorter string in the pair. For example, suppose that an ArrayList called list contains the following values: {"four", "score", "and", "seven", "years", "ago"} In the first pair, "four" and "score", the shorter string is "four". In the second pair, "and" and "seven", the shorter string is "and". In the third pair, "years" and "ago", the shorter string is "ago". Therefore, the call: removeShorterStrings(list); should remove these shorter strings, leaving the list as follows: "score", "seven", "years". If there is a tie (both strings have the same length), your method should remove the first string in the pair. If there is an odd number of strings in the list, the final value should be kept in the list.

```
public void removeShorterStrings(ArrayList<String> list) {
    for (int i = 0; i < list.size() - 1; i++) {
        String first = list.get(i);
        String second = list.get(i + 1);
        if (first.length() <= second.length()) {
            list.remove(i);
        } else {
            list.remove(i + 1);
        }
    }
}
```

Exercise 10.15: filterRange

Write a method filterRange that accepts an ArrayList of integers and two integer values *min* and *max* as parameters and removes all elements whose values are in the range *min* through *max* (inclusive) from the list. For example, if a variable called list stores the values:

[4, 7, 9, 2, 7, 7, 5, 3, 5, 1, 7, 8, 6, 7]

The call of filterRange(list, 5, 7); should remove all values between 5 and 7, therefore it should change the list to store [4, 9, 2, 3, 1, 8]. If no elements in range *min-max* are found in the list, the list's contents are unchanged. If an empty list is passed, the list remains empty. You may assume that the list is not null.

```
public static void filterRange(ArrayList<Integer> list, int min, int max) {
    for (int i = 0; i < list.size(); i++) {
        if (list.get(i) >= min && list.get(i) <= max) {
            list.remove(i);
            i--;
        }
    }
}
```

Exercise 10.17: interleave

Write a method called interleave that accepts two ArrayLists of integers *a1* and *a2* as parameters and inserts the elements of *a2* into *a1* at alternating indexes. If the lists are of unequal length, the remaining elements of the longer list are left at the end of *a1*. For example, if *a1* stores [10, 20, 30] and *a2* stores [4, 5, 6, 7, 8], the call of interleave(a1, a2); should change *a1* to store [10, 4, 20, 5, 30, 6, 7, 8]. If *a1* had stored [10, 20, 30, 40, 50] and *a2* had stored [6, 7, 8], the call of interleave(a1, a2); would change *a1* to store [10, 6, 20, 7, 30, 8, 40, 50].

```
public static void interleave(ArrayList<Integer> a1, ArrayList<Integer> a2) {
    for (int i = 0; i < a2.size(); i++) {
        int index = Math.min(a1.size(), 2 * i + 1);
        a1.add(index, a2.get(i));
    }
}
```

Exercise 10.18: ComparablePoint

Suppose that a class Point2D has been defined for storing a 2-dimensional point with x and y coordinates (both as doubles). (In our section handout, the class was called Point, but we have renamed it here because Practice-It uses a class named Point for other purposes.) The class includes the following members:

Name	Description
private double x	the x coordinate
private double y	the y coordinate
public Point2D()	constructs the point (0, 0)
public Point2D(double x, double y)	constructs a point with the given coordinates
public void setLocation(double x, double y)	sets the coordinates to the given values
public double getX()	returns the x coordinate
public double getY()	returns the y coordinate
public String toString()	returns a String in standard "(x, y)" notation
public double distance(Point2D other)	returns the distance from another point

Your task is to modify the class to be Comparable by adding an appropriate compareTo method. Points should be compared relative to their distance from the origin, with points closer to the origin considered "less" than points farther from it. The distance between two points is defined as the square root of the sum of the squares of the differences between the x and y coordinates.

```
public int compareTo(Point2D p) {
   double distance1 = Math.sqrt(x * x + y * y);
   double distance2 = Math.sqrt(p.x * p.x + p.y * p.y);
   if (distance1 - distance2 < 0) {
      return -1;
   } else if (distance1 - distance2 > 0) {
      return 1;
   } else {
      return 0;
   }
}
```

Exercise 10.20: ComparableCalenderDate
Suppose that a class CalendarDate has been defined for storing a calendar date with month, day and year components. (In our section handout, the class was called Date, but we have renamed it here because Practice-It uses a class named Date for other purposes.) The class includes the following members:

Name	Description
private int year	the year component
private int month	the month component
private int day	the day component
public CalendarDate(int year, int month, int day)	constructs a date with given year, month, day
public int getYear()	returns the year component
public int getMonth()	returns the month component
public int getDay()	returns the day component
public String toString()	returns the date in *yyyy/mm/dd* format

Your task is to modify the class to be Comparable by adding an appropriate compareTo method. Dates that occur chronologically earlier should be considered "less" than dates that occur later.

You may assume that dates are constructed with appropriate values: Months will be between 1 and 12, days will be between 1 and 31 and years will be four-digit numbers. Assume that the toString method produces a standard format with two digits for the month and two digits for the day, including leading zeros if necessary. For example, January 7th, 2005, would return the string "2005/01/07".

```
public int compareTo(CalendarDate date) {
   if (year == date.year && month == date.month && day == date.day) {
      return 0;
   } else if (year == date.year && month == date.month && day > date.day) {
      return 1;
   } else if (year == date.year && month > date.month) {
      return 1;
   } else if (year > date.year) {
      return 1;
   } else {
      return -1;
   }
}
```

Chapter 11: Java Collections Framework

Exercise 11.2: alternate

Write a method called alternate that accepts two Lists of integers as its parameters and returns a new List containing alternating elements from the two lists, in the following order:

- First element from first list
- First element from second list
- Second element from first list
- Second element from second list
- Third element from first list
- Third element from second list
- …

If the lists do not contain the same number of elements, the remaining elements from the longer list should be placed consecutively at the end. For example, for a first list of (1, 2, 3, 4, 5) and a second list of (6, 7, 8, 9, 10, 11, 12), a call of alternate(list1, list2) should return a list containing (1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 11, 12). Do not modify the parameter lists passed in.

```
public static List<Integer> alternate(List<Integer> list1, List<Integer> list2) {
   List<Integer> result = new ArrayList<Integer>();
   Iterator<Integer> i1 = list1.iterator();
   Iterator<Integer> i2 = list2.iterator();
   while (i1.hasNext() || i2.hasNext()) {
      if (i1.hasNext()) {
         result.add(i1.next());
      }
      if (i2.hasNext()) {
         result.add(i2.next());
      }
   }
   return result;
}
```

Exercise 11.3: removeInRange

Write a method called removeInRange that accepts four parameters: a List of integers, an element value, a starting index, and an ending index. The method's behavior is to remove all occurrences of the given element that appear in the list between the starting index (inclusive) and the ending index (exclusive). Other values and occurrences of the given value that appear outside the given index range are not affected.

For example, for the list (0, 0, 2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0, 16), a call of removeInRange(list, 0, 5, 13) should produce the list (0, 0, 2, 0, 4, 6, 8, 10, 12, 0, 14, 0, 16). Notice that the zeros located at indices between 5 inclusive and 13 exclusive in the original list (before any modifications were made) have been removed.

```
public static void removeInRange(List<Integer> list, int value, int min, int max) {
   Iterator<Integer> itr = list.iterator();
   for (int i = 0; i < min; i++) {
      itr.next();
   }
   for (int i = min; i < max; i++) {
      if (itr.next() == value) {
         itr.remove();
      }
   }
}
```

Exercise 11.6: countUnique
Write a method countUnique that takes a List of integers as a parameter and returns the number of unique integer values in the list. Use a Set as auxiliary storage to help you solve this problem.

For example, if a list contains the values [3, 7, 3, -1, 2, 3, 7, 2, 15, 15], your method should return 5. The empty list contains 0 unique values.

```
public static int countUnique(List<Integer> list) {
   Set<Integer> set = new HashSet<Integer>();
   for (int value : list) {
      set.add(value);
   }
   return set.size();
}
```

Exercise 11.7: countCommon
Write a method countCommon that takes two Lists of integers as parameters and returns the number of unique integers that occur in both lists. Use one or more Sets as storage to help you solve this problem.

For example, if one list contains the values [3, 7, 3, -1, 2, 3, 7, 2, 15, 15] and the other list contains the values [-5, 15, 2, -1, 7, 15, 36], your method should return 4 (because the elements -1, 2, 7, and 15 occur in both lists).

```
public static int countCommon(List<Integer> list1, List<Integer> list2) {
    Set<Integer> set1 = new HashSet<Integer>(list1);
    Set<Integer> set2 = new HashSet<Integer>(list2);
    int common = 0;
    for (int value : set1) {
        if (set2.contains(value)) {
            common++;
        }
    }
    return common;
}
```

Exercise 11.8: maxLength
Write a method maxLength that takes a Set of strings as a parameter and that returns the length of the longest string in the set. If your method is passed an empty set, it should return 0.

```
public static int maxLength(Set<String> set) {
    int max = 0;
    for (String s : set) {
        max = Math.max(max, s.length());
    }
    return max;
}
```

Exercise 11.9 hasOdd
Write a method hasOdd that takes a Set of integers as a parameter and that returns true if the set contains at least one odd integer, and false otherwise. If passed the empty set, your method should return false.

```
public static boolean hasOdd(Set<Integer> set) {
    for (int value : set) {
        if (value % 2 != 0) {
            return true;
        }
    }
    return false;
}
```

Exercise 11.10: removeEvenLength
Write a method removeEvenLength that takes a Set of strings as a parameter and that removes all of the strings of even length from the set. For example, if your method is passed a set containing the following elements:

```
["foo", "buzz", "bar", "fork", "bort", "spoon", "!", "dude"]
```

Your method should modify the set to store the following elements (the order of the elements does not matter):

["foo", "bar", "spoon", "!"]

```
public static void removeEvenLength(Set<String> set) {
   Iterator<String> itr = set.iterator();
   while (itr.hasNext()) {
      String s = itr.next();
      if (s.length() % 2 == 0) {
         itr.remove();
      }
   }
}
```

Exercise 11.12: contains3
Write a method contains3 that accepts a List of strings as a parameter and returns true if any single string occurs at least 3 times in the list, and false otherwise. Use a map as auxiliary storage.

```
public static boolean contains3(List<String> list) {
   Map<String, Integer> counts = new HashMap<String, Integer>();
   for (String value : list) {
      if (counts.containsKey(value)) {
         int count = counts.get(value);
         count++;
         counts.put(value, count);
         if (count >= 3) {
            return true;
         }
      } else {
         counts.put(value, 1);
      }
   }
   return false;
}
```

Exercise 11.13: isUnique
Write a method isUnique that accepts a Map from strings to strings as a parameter and returns true if no two keys map to the same value (and false if any two or more keys do map to the same value). For example, calling your method on the following map would return true:

{Marty=Stepp, Stuart=Reges, Jessica=Miller, Amanda=Camp, Hal=Perkins}

Calling it on the following map would return false, because of two mappings for Perkins and Reges:

{Kendrick=Perkins, Stuart=Reges, Jessica=Miller, Bruce=Reges, Hal=Perkins}

The empty map is considered to be unique, so your method should return true if passed an empty map.

```
public static boolean isUnique(Map<String, String> map) {
   Set<String> values = new HashSet();
   for (String value : map.values()) {
      if (values.contains(value)) {
         return false;   // duplicate
      } else {
         values.add(value);
      }
   }
   return true;
}
```

Exercise 11.14: intersect

Write a method intersect that takes two Maps of strings to integers as parameters and that returns a new map whose contents are the intersection of the two. The intersection of two maps is defined here as the set of keys and values that exist in both maps. So if some key K maps to value V in both the first and second map, include it in your result. If K does not exist as a key in both maps, or if K does not map to the same value V in both maps, exclude that pair from your result. For example, consider the following two maps:

{Janet=87, Logan=62, Whitaker=46, Alyssa=100, Stefanie=80, Jeff=88, Kim=52, Sylvia=95}
{Logan=62, Kim=52, Whitaker=52, Jeff=88, Stefanie=80, Brian=60, Lisa=83, Sylvia=87}

Calling your method on the preceding maps would return the following new map (the order of the key/value pairs does not matter):

{Logan=62, Stefanie=80, Jeff=88, Kim=52}

```
public static Map<String, Integer> intersect(Map<String, Integer> map1, Map<String, Integer>
map2) {
   Map<String, Integer> result = new TreeMap<String, Integer>();
   for (String key : map1.keySet()) {
      int value = map1.get(key);
      if (map2.containsKey(key) && value == map2.get(key)) {
         result.put(key, value);
```

```
        }
    }
    return result;
}
```

Exercise 11.15: maxOccurrences

Write a method maxOccurrences that accepts a List of integers as a parameter and returns the number of times the most frequently occurring integer (the "mode") occurs in the list. Solve this problem using a Map as auxiliary storage. If the list is empty, return 0.

```
public static int maxOccurrences(List<Integer> list) {
    Map<Integer, Integer> counts = new HashMap<Integer, Integer>();
    for (int value : list) {
        if (counts.containsKey(value)) {
            counts.put(value, counts.get(value) + 1);
        } else {
            counts.put(value, 1);
        }
    }

    int max = 0;
    for (int count : counts.values()) {
        max = Math.max(max, count);
    }
    return max;
}
```

Exercise 11.18: reverse

Write a method reverse that accepts a Map from integers to strings as a parameter and returns a new Map of strings to integers that is the original's "reverse". The reverse of a map is defined here to be a new map that uses the values from the original as its keys and the keys from the original as its values. Since a map's values need not be unique but its keys must be, it is acceptable to have any of the original keys as the value in the result. In other words, if the original map has pairs (*k1, v*) and (*k2, v*), the new map must contain either the pair (*v, k1*) or (*v, k2*).

For example, for the following map:

{42=Marty, 81=Sue, 17=Ed, 31=Dave, 56=Ed, 3=Marty, 29=Ed}

Your method could return the following new map (the order of the key/value pairs does not matter):

{Marty=3, Sue=81, Ed=29, Dave=31}

```
public static Map<String, Integer> reverse(Map<Integer, String> map) {
    Map<String, Integer> result = new HashMap<String, Integer>();
    for (Integer key : map.keySet()) {
        String value = map.get(key);
        result.put(value, key);
    }
    return result;
}
```

Exercise 11.19: rarest

Write a method rarest that accepts a map whose keys are strings and whose values are integers as a parameter and returns the integer value that occurs the fewest times in the map. If there is a tie, return the smaller integer value. If the map is empty, throw an exception.

For example, suppose the map contains mappings from students' names (strings) to their ages (integers). Your method would return the least frequently occurring age. Consider a map variable m containing the following key/value pairs:

{Alyssa=22, Char=25, Dan=25, Jeff=20, Kasey=20, Kim=20, Mogran=25, Ryan=25, Stef=22}
Three people are age 20 (Jeff, Kasey, and Kim), two people are age 22 (Alyssa and Stef), and four people are age 25 (Char, Dan, Mogran, and Ryan). So a call of rarest(m) returns 22 because only two people are that age.

If there is a tie (two or more rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair of Kelly=22 to the map above, there would now be a tie of three people of age 20 (Jeff, Kasey, Kim) and three people of age 22 (Alyssa, Kelly, Stef). So a call of rarest(m) would now return 20 because 20 is the smaller of the rarest values.

```
public static int rarest(Map<String, Integer> m) {
  if (m == null || m.isEmpty()) {
    throw new IllegalArgumentException();
  }
  Map<Integer, Integer> counts = new TreeMap<Integer, Integer>();
  for (String name : m.keySet()) {
    int age = m.get(name);
    if (counts.containsKey(age)) {
      counts.put(age, counts.get(age) + 1);
    } else {
      counts.put(age, 1);
    }
    }

    int minCount = m.size() + 1;
    int rareAge = -1;
    for (int age : counts.keySet()) {
        int count = counts.get(age);
        if (count < minCount) {
            minCount = count;
            rareAge = age;
        }
    }

    return rareAge;
}
```

Chapter 12: Recursion

Exercise 12.1: starString

Write a method starString that accepts an integer parameter *n* and returns a string of stars (asterisks) 2^n long (i.e., 2 to the n^{th} power). For example:

Call	Output	Reason
starString(0);	"*"	$2^0 = 1$
starString(1);	"**"	$2^1 = 2$
starString(2);	"****"	$2^2 = 4$
starString(3);	"********"	$2^3 = 8$
starString(4);	"****************"	$2^4 = 16$

You should throw an IllegalArgumentException if passed a value less than 0.

```
public String starString(int n) {
   if (n < 0) {
      throw new IllegalArgumentException();
   } else if (n == 0) {
      return "*";
   } else {
      return starString(n - 1) + starString(n - 1);
   }
}
```

Exercise 12.2: writeNums

Write a method writeNums that accepts an integer parameter *n* and prints the first *n* integers starting with 1 in sequential order, separated by commas. For example, the following calls produce the following output:

Call	Output
writeNums(5);	1, 2, 3, 4, 5
writeNums(12);	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Your method should throw an IllegalArgumentException if passed a value less than 1. Note that the output does not advance to the next line.

```
public void writeNums(int n) {
   if (n < 1) {
       throw new IllegalArgumentException();
   } else if (n == 1) {
       System.out.print(1);
   } else {
       writeNums(n - 1);
       System.out.print(", " + n);
   }
}
```

Exercise 12.3: writeSequence

Write a method writeSequence that accepts an integer *n* as a parameter and prints a symmetric sequence of *n* numbers with descending integers ending in 1 followed by ascending integers beginning with 1, as in the table below:

Call	Output
writeSequence(1);	1
writeSequence(2);	1 1
writeSequence(3);	2 1 2
writeSequence(4);	2 1 1 2
writeSequence(5);	3 2 1 2 3
writeSequence(6);	3 2 1 1 2 3
writeSequence(7);	4 3 2 1 2 3 4
writeSequence(8);	4 3 2 1 1 2 3 4
writeSequence(9);	5 4 3 2 1 2 3 4 5
writeSequence(10);	5 4 3 2 1 1 2 3 4 5

Notice that for odd numbers the sequence has a single 1 in the middle while for even values it has two 1s in the middle.

Your method should throw an IllegalArgumentException if passed a value less than 1. A client using this method would have to call println to complete the line of output.

```
public void writeSequence(int n) {
    if (n < 1) {
        throw new IllegalArgumentException();
    } else if (n == 1) {
        System.out.print(1);
    } else if (n == 2) {
        System.out.print("1 1");
    } else {
        int number = (n + 1) / 2;
        System.out.print(number + " ");
        writeSequence(n - 2);
        System.out.print(" " + number);
    }
}
```

Exercise 12.6: writeSquares

Write a method writeSquares that accepts an integer parameter *n* and prints the first *n* squares separated by commas, with the odd squares in descending order followed by the even squares in ascending order. The following table shows several calls to the method and their expected output:

Call	**Valued Returned**
writeSquares(5);	25, 9, 1, 4, 16
writeSquares(1);	1
writeSquares(8);	49, 25, 9, 1, 4, 16, 36, 64

Your method should throw an IllegalArgumentException if passed a value less than 1. Note that the output does not advance onto the next line.

```
public void writeSquares(int n) {
    if (n < 1) {
        throw new IllegalArgumentException();
    } else if (n == 1) {
        System.out.print(1);
    } else if (n % 2 == 1) {
        System.out.print(n * n + ", ");
        writeSquares(n - 1);
    } else {
        writeSquares(n - 1);
        System.out.print(", " + n * n);
    }
}
```

Exercise 12.7: writeChars

Write a method writeChars that accepts an integer parameter *n* and that prints out *n* characters as follows. The middle character of the output should always be an asterisk ("*"). If you are asked to write out an even number of characters, then there will be two asterisks in the middle ("**"). Before the asterisk(s) you should write out less-than characters ("<"). After the asterisk(s) you should write out greater-than characters (">"). For example, the following calls produce the following output:

Call	Output
writeChars(1);	*
writeChars(2);	**
writeChars(3);	<*>
writeChars(4);	<**>
writeChars(5);	<<*>>
writeChars(6);	<<**>>
writeChars(7);	<<<*>>>
writeChars(8);	<<<**>>>

Your method should throw an IllegalArgumentException if passed a value less than 1. Note that the output does not advance to the next line.

```
public void writeChars(int n) {
   if (n < 1) {
      throw new IllegalArgumentException();
   } else if (n == 1) {
      System.out.print("*");
   } else if (n == 2) {
      System.out.print("**");
   } else {
      System.out.print("<");
      writeChars(n - 2);
      System.out.print(">");
   }
}
```

Exercise 12.8: multiplyEvens

Write a method multiplyEvens that returns the product of the first n even integers. For example:

Call	Output	Reason
multiplyEvens(1);	2	2 = 2
multiplyEvens(2);	8	2 * 4 = 8

multiplyEvens(3);	48	2 * 4 * 6 = 48
multiplyEvens(4);	384	2 * 4 * 6 * 8 = 384

You should throw an IllegalArgumentException if passed a value less than or equal to 0.

```
public int multiplyEvens(int n) {
   if (n < 1) {
      throw new IllegalArgumentException();
   } else if (n == 1) {
      return 2;
   } else {
      return 2 * n * multiplyEvens(n - 1);
   }
}
```

Exercise 12.9: sumTo

Write a method sumTo that accepts an integer parameter *n* and returns the sum of the first n reciprocals. In other words:

sumTo(n) returns: 1 + 1/2 + 1/3 + 1/4 + ... + 1/*n*

For example, the call of sumTo(2) should return 1.5. The method should return 0.0 if passed the value 0 and should throw an IllegalArgumentException if passed a value less than 0.

```
public double sumTo(int n) {
   if (n < 0) {
      throw new IllegalArgumentException();
   } else if (n == 0) {
      return 0.0;
   } else {
      return sumTo(n - 1) + 1.0 / n;
   }
}
```

Exercise 12.10: digitmatch

Robert Baxter (on 2016/09/08)

Write a recursive method digitMatch that accepts two non-negative integers as parameters and that returns the number of digits that match between them. Two digits match if they are equal and have the same position relative to the end of the number (i.e. starting with the ones digit). In other words, the method should compare the last digits of each number, the second-to-last digits of each number, the third-to-last digits of each number, and so forth, counting how many pairs match. For example, for the call of digitMatch(1072503891, 62530841), the method would compare as follows:

```
1 0 7 2 5 0 3 8 9 1
    | | | | | | | |
    6 2 5 3 0 8 4 1
```

The method should return 4 in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). Below are more examples:

Call	Value Returned
digitMatch(38, 34)	1
digitMatch(5, 5552)	0
digitMatch(892, 892)	3
digitMatch(298892, 7892)	3
digitMatch(380, 0)	1
digitMatch(123456, 654321)	0
digitMatch(1234567, 67)	2

Your method should throw an IllegalArgumentException if either of the two parameters is negative. You are not allowed to construct any structured objects other than Strings (no array, List, Scanner, etc.) and you may not use any loops to solve this problem; you must use recursion.

```
public static int digitMatch(int x, int y) {
    if (x < 0 || y < 0) {
        throw new IllegalArgumentException();
    } else if (x < 10 || y < 10) {
        if (x % 10 == y % 10) {
            return 1;
        } else {
            return 0;
        }
    } else if (x % 10 == y % 10) {
        return 1 + digitMatch(x / 10, y / 10);
    } else {
        return digitMatch(x / 10, y / 10);
    }
}
```

Exercise 12.11: repeat

Write a recursive method repeat that accepts a string *s* and an integer *n* as parameters and that returns a String consisting of *n* copies of *s*. For example:

Call	Value Returned
repeat("hello", 3)	"hellohellohello"
repeat("this is fun", 1)	"this is fun"
repeat("wow", 0)	""
repeat("hi ho! ", 5)	"hi ho! hi ho! hi ho! hi ho! hi ho! "

You should solve this problem by concatenating String objects using
the + operator. String concatenation is an expensive operation, so it is best to minimize the number of concatenation operations you perform. For example, for the call repeat("foo", 500), it would be inefficient to perform 500 different concatenation operations to obtain the result. Most of the credit will be awarded on the correctness of your solution independent of efficiency. The remaining credit will be awarded based on your ability to minimize the number of concatenation operations performed.

Your method should throw an IllegalArgumentException if passed any negative value for *n*. You are not allowed to construct any structured objects other than Strings (no array, List, Scanner, etc.) and you may not use any loops to solve this problem; you must use recursion.

```
public static String repeat(String s, int n) {
   if(n < 0) {
      throw new IllegalArgumentException();
   } else if(n == 0) {
      return "";
   } else if (n == 1) {
      return s;
   } else if (n % 2 == 0) {
      String temp = repeat(s, n / 2);
      return temp + temp;
   } else {
      return s + repeat(s, n - 1);
   }
}
```

Exercise 12.12: isReverse

Write a recursive method isReverse that accepts two strings as a parameter and returns true if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalization), and false otherwise. For example, the string "hello" backwards is "olleh", so a call of isReverse("hello", "olleh") would return true. Since the method is case-insensitive, you would also get a true result from a call of isReverse("Hello", "oLLEh"). The empty string, as well as any one-letter string, is considered to be its own reverse. The string could contain characters other than letters, such as numbers, spaces, or other punctuation; you should treat these like any other character. The key aspect is that the first string has the same sequence of characters as the second string, but in the opposite order, ignoring case. The table below shows more examples:

Call	Value Returned
isReverse("CSE143", "341esc")	true
isReverse("Madam", "MaDAm")	true
isReverse("Q", "Q")	true
isReverse("", "")	true
isReverse("e via n", "N aIv E")	true
isReverse("Go! Go", "OG !OG")	true
isReverse("Obama", "McCain")	false
isReverse("banana", "nanaba")	false
isReverse("hello!!", "olleh")	false
isReverse("", "x")	false
isReverse("madam I", "i m adam")	false
isReverse("ok", "oko")	false

You may assume that the strings passed are not null. You are not allowed to construct any structured objects other than Strings (no array, List, Scanner, etc.) and you may not use any loops to solve this problem; you must use recursion. If you like, you may declare other methods to help you solve this problem, subject to the previous rules.

```
public static boolean isReverse(String s1, String s2) {
   if (s1.length() == 0 && s2.length() == 0) {
      return true;
   } else if (s1.length() == 0 || s2.length() == 0) {
      return false;  // not same length
   } else {
      String s1first = s1.substring(0, 1);
      String s2last = s2.substring(s2.length() - 1);
```

```
        return s1first.equalsIgnoreCase(s2last) &&
              isReverse(s1.substring(1), s2.substring(0, s2.length() - 1));
    }
}
```

Exercise 12.13: indexOf
Robert Baxter (on 2016/09/08)

Write a recursive method indexOf that accepts two Strings as parameters and that returns the starting index of the first occurrence of the second String inside the first String (or -1 if not found). The table below lists several calls to your method and their expected return values. Notice that case matters, as in the last example that returns -1.

Call	Value Returned
indexOf("Barack Obama", "Bar")	0
indexOf("Barack Obama", "ck")	4
indexOf("Barack Obama", "a")	1
indexOf("Barack Obama", "McCain")	-1
indexOf("Barack Obama", "BAR")	-1

Strings have an indexOf method, but you are not allowed to call it. You are limited to these methods:

Method	Description
equals(String *other*)	returns true if the two Strings contain the same characters
length()	returns the int number of characters in the String

substring(int *fromIndex*, int *toIndex*)
substring(int *fromIndex*)
	returns a new String containing the characters from this String from *fromIndex* (inclusive) to *toIndex* (exclusive), or to the end of the String if *toIndex* is omitted

You are not allowed to construct any structured objects other than Strings (no array, List, Scanner, etc.) and you may not use any loops to solve this problem; you must use recursion.

```
public static int indexOf(String source, String target) {
    if(target.length() > source.length()) {
        return -1;
    } else if(source.substring(0, target.length()).equals(target)) {
        return 0;
    } else {
        int pos = indexOf(source.substring(1), target);
        if(pos == -1) {
           return -1;
        } else {
           return pos + 1;
        }
    }
}
```

Exercise 12.14: evenDigits

Write a method evenDigits that accepts an integer parameter *n* and that returns the integer formed by removing the odd digits from *n*. The following table shows several calls and their expected return values:

Call	Valued Returned
evenDigits(8342116);	8426
evenDigits(4109);	40
evenDigits(8);	8
evenDigits(-34512);	-42
evenDigits(-163505);	-60
evenDigits(3052);	2
evenDigits(7010496);	46
evenDigits(35179);	0
evenDigits(5307);	0
evenDigits(7);	0

If a negative number with even digits other than 0 is passed to the method, the result should also be negative, as shown above when -34512 is passed. Leading zeros in the result should be ignored and if there are no even digits other than 0 in the number, the method should return 0, as shown in the last three outputs.

```
public int evenDigits(int n) {
        if (n < 0) {
                return -evenDigits(-n);
        } else if (n == 0) {
                return 0;
        } else if (n % 2 == 0) {
                return 10 * evenDigits(n / 10) + n % 10;
        } else {
            return evenDigits(n / 10);
        }
}
```

Exercise 12.18: waysToClimb
In this problem, the scenario we are evaluating is the following: You're standing at the base of a staircase and are heading to the top. A small stride will move up one stair, and a large stride advances two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. For example, a staircase of three steps can be climbed in three different ways: three small strides, one small stride followed by one large stride, or one large followed by one small.

Write a recursive method waysToClimb that takes a non-negative integer value representing a number of stairs and prints each unique way to climb a staircase of that height, taking strides of one or two stairs at a time. Your method should output each way to climb the stairs on its own line, using a 1 to indicate a small stride of 1 stair, and a 2 to indicate a large stride of 2 stairs. For example, the call of waysToClimb(3) should produce the following output:

```
[1, 1, 1]
[1, 2]
[2, 1]
```

The call of waysToClimb(4) should produce the following output:

```
[1, 1, 1, 1]
[1, 1, 2]
[1, 2, 1]
[2, 1, 1]
[2, 2]
```

The order in which you output the possible ways to climb the stairs is not important, so long as you list the right overall set of ways. There are no ways to climb zero stairs, so your method should produce no output if 0 is passed. Do not use any loops in solving this problem.

```
public static void waysToClimb(int n) {
   waysToClimb(n, 0, "[");
}

private static void waysToClimb(int n, int position, String result) {
    if (n == position) {
        int index = result.lastIndexOf(",");
        if (index != -1) {
            result = result.substring(0, index) + "]";
            System.out.println(result);
        }
    } else if (n > position) {
        waysToClimb(n, position + 1, result + "1, ");
        waysToClimb(n, position + 2, result + "2, ");
    }
}
```

Exercise 12.19: countBinary

Write a method countBinary that accepts an integer n as a parameter and that prints all binary numbers that have n digits in ascending order, printing each value on a separate line. All n digits should be shown for all numbers, including leading zeros if necessary. You may assume that n is non-negative. If n is 0, a blank line of output should be produced. Do not use a loop in your solution; implement it recursively.

Call	Output

countBinary(1);	
	```
	0
	1
	```

countBinary(2);	
	```
	00
	01
	10
	11
	```

countBinary(3);	
	```
	000
	001
	010
	011
	100
	101
	110
	111
	```

Hint: It may help to define a private helper method that accepts different parameters than the original method. In particular, consider building up a set of characters as a String for eventual printing.

```
public static void countBinary(int n) {
   countBinary(n, "");
}

private static void countBinary(int digitsLeft, String s) {
   if (digitsLeft == 0) {
      System.out.println(s);
   } else {
      countBinary(digitsLeft - 1, s + "0");
      countBinary(digitsLeft - 1, s + "1");
   }
}
```

Exercise 12.20: subsets

Write a method subsets that uses recursive backtracking to find every possible sub-list of a given list. A sub-list of a list *L* contains 0 or more of *L*'s elements. Your method should accept a List of strings as its parameter and print every sub-list that could be created from elements of that list, one per line. For example, suppose a variable called list stores the following elements:

```
[Janet, Robert, Morgan, Char]
```

The call of subsets(list); would produce output such as the following:

```
[Janet, Robert, Morgan, Char]
[Janet, Robert, Morgan]
[Janet, Robert, Char]
[Janet, Robert]
[Janet, Morgan, Char]
[Janet, Morgan]
[Janet, Char]
[Janet]
[Robert, Morgan, Char]
[Robert, Morgan]
[Robert, Char]
[Robert]
[Morgan, Char]
[Morgan]
[Char]
[]
```

The order in which you show the sub-lists does not matter, but the order of the elements of each sub-list DOES matter (Note: this requirement is a limitation of Practice-It testing code. The textbook version of the problem would accept an answer with the elements in any order). The key thing is that your method should produce the correct overall set of sub-lists as its output. Notice that the empty list is considered one of these sub-lists. You may assume that the list passed to your method is not null and that the list contains no duplicates. Do not use any loops in solving this problem.

Hint: This problem is somewhat similar to the permutations problem. Consider each element and try to generate all sub-lists that do include it, as well as all sub-lists that do not include it.

It can be hard to see a pattern from looking at the lines of output. But notice that the first 8 of 16 total lines of output constitute all the sets that include Janet, and the last 8 lines are the sets that do not have her as a member. Within either of those groups of 8 lines, the first 4 of them are all the sets that include Robert, and the last 4 lines are the ones that do not include him. Within a clump of 4, the first 2 are the ones including Morgan, and the last 2 are the ones that do not include Morgan. And so on. Once again, you do not have to match this exact order, but looking at it can help with figuring out the patterns and the recursion.

```
// Prints all sub-lists of the given list of Strings.
// Precondition: elements != null and elements contains no duplicates
public static void subsets(List<String> elements) {
   List<String> chosen = new ArrayList<String>();
   explore(elements, chosen);
}

// Private recursive helper to explore all sub-lists of the given list of
// elements, assuming the given list of strings have already been chosen.
private static void explore(List<String> elements, List<String> chosen) {
   if (elements.isEmpty()) {
      System.out.println(chosen);   // base case; nothing left to choose
   } else {
      String first = elements.remove(0);   // make a choice: 1st element

      // two explorations: one with this first element, one without
      chosen.add(first);
      explore(elements, chosen);
      chosen.remove(chosen.size() - 1);
      explore(elements, chosen);

      elements.add(0, first);     // backtrack!  put 1st element back
   }
}
```

Exercise 12.21: maxSum

Write a recursive method maxSum that accepts a list of integers L and an integer limit n as its parameters and uses backtracking to find the maximum sum that can be generated by adding elements of L that does not exceed n. For example, if you are given the list of integers [7, 30, 8, 22, 6, 1, 14] and the limit of 19, the maximum sum that can be generated that does not exceed is 16, achieved by adding 7, 8, and 1. If the list L is empty, or if the limit is not a positive integer, or all of L's values exceed the limit, return 0.

Each index's element in the list can be added to the sum only once, but the same number value might occur more than once in a list, in which case each occurrence might be added to the sum. For example, if the list is [6, 2, 1] you may use up to one 6 in the sum, but if the list is [6, 2, 6, 1] you may use up to two sixes.

Here are several example calls to your method and their expected return values:

List L	**Limit n**	**maxSum(L, n) returns**
[7, 30, 8, 22, 6, 1, 14]	19	16
[5, 30, 15, 13, 8]	42	41
[30, 15, 20]	40	35
[6, 2, 6, 9, 1]	30	24
[11, 5, 3, 7, 2]	14	14
[10, 20, 30]	7	0
[10, 20, 30]	20	20
[]	10	0

You may assume that all values in the list are non-negative. Your method may alter the contents of the list L as it executes, but L should be restored to its original state before your method returns. Do not use any loops in solving this problem.

```
public static int maxSum(List<Integer> numbers, int limit) {
    if (limit <= 0 || numbers.isEmpty()) {
        return 0;
    } else {
        int first = numbers.get(0);
        numbers.remove(0);
        int max;
        if (first > limit) {
            max = maxSum(numbers, limit);
        } else {
            int with    = first + maxSum(numbers, limit - first);
            int without = maxSum(numbers, limit);
            max = Math.max(with, without);
        }

        numbers.add(0, first);
        return max;
    }
}
```

Exercise 12.22: printSquares

Write a method printSquares that uses recursive backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example, the call of printSquares(200); should produce the following output:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
2^2 + 4^2 + 6^2 + 12^2
2^2 + 14^2
3^2 + 5^2 + 6^2 + 7^2 + 9^2
6^2 + 8^2 + 10^2
```

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your method should produce no output. Keep in mind that the sum has to be formed with unique integers. Otherwise you could always find a solution by adding 1^2 together until you got to whatever number you are working with.

As with any backtracking problem, this one amounts to a set of choices, one for each integer whose square might or might not be part of your sum. In many of our backtracking problems we store the choices in some kind of collection. In this problem you can instead generate the choices by doing a for loop over an appropriate range of numbers. Note that the maximum possible integer that can be part of a sum of squares for an integer *n* is the square root of *n*.

Like with other backtracking problems, you still need to keep track of which choices you have made at any given moment. In this case, the choices you have made consist of some group of integers whose squares may be part of a sum that will add up to *n*. Represent these chosen integers as an appropriate collection where you add the integer *i* to the collection to consider it as part of an answer. If you ever create such a collection whose values squared add up to *n*, you have found a sum that should be printed.

To help you solve this problem, assume there already exists a method printHelper that accepts any Java collection of integers (such as a list, set, stack, queue, etc.) and prints the collection's elements in order. For example, if a set s stores the elements [1, 4, 8, 11], the call of printHelper(s); would produce the following output:

```
1^2 + 4^2 + 8^2 + 11^2
```

```
// Prints all ways to express n as a sum of squares of unique integers.
// Precondition: n >= 0
public static void printSquares(int n) {
   Set<Integer> chosen = new TreeSet<Integer>();
   explore(n, 1, chosen);
}

// Explore all ways to form n as a sum of squares of integers starting
// with the given min and storing the chosen results in the given set.
private static void explore(int n, int min, Set<Integer> chosen) {
   if (n == 0) {
      printHelper(chosen);   // base case: sum has reached n
   } else {
      // recursive case: try all combinations of every integer
      int max = (int) Math.sqrt(n);   // valid choices go up to sqrt(n)

      for (int i = min; i <= max; i++) {
         // try all combinations that include the square of this integer
         chosen.add(i);
         explore(n - (i * i), i + 1, chosen);
         chosen.remove(i);                    // backtrack
      }
   }
}
```

Chapter 14: Searching and Sorting

Exercise 14.1: splitStack

Write a method splitStack that takes a stack of integers as a parameter and splits it into negatives and non-negatives. The numbers in the stack should be rearranged so that all the negatives appear on the bottom of the stack and all the non-negatives appear on the top. In other words, if after this method is called you were to pop numbers off the stack, you would first get all the nonnegative numbers and then get all the negative numbers. It does not matter what order the numbers appear in as long as all the negatives appear lower in the stack than all the non-negatives. You may use a single queue as auxiliary storage.

```
public void splitStack(Stack<Integer> s) {
   Queue<Integer> q = new LinkedList<Integer>();

   // transfer all elements from stack to queue
   int oldLength = s.size();
   while (!s.isEmpty()) {
      q.add(s.pop());
   }

   // transfer negatives from queue to stack
   for (int i = 1; i <= oldLength; i++) {
      int n = q.remove();
      if (n < 0) {
         s.push(n);
      } else {
         q.add(n);
      }
   }

   // transfer nonnegatives from queue to stack
   while (!q.isEmpty()) {
      s.push(q.remove());
   }
}
```

Exercise 14.2: stutter

Write a method stutter that takes a stack of integers as a parameter and replaces every value in the stack with two occurrences of that value. For example, suppose the stack stores these values:

bottom [3, 7, 1, 14, 9] top

Then the stack should store these values after the method terminates:

bottom [3, 3, 7, 7, 1, 1, 14, 14, 9, 9] top

Notice that you must preserve the original order. In the original list the 9 was at the top and would have been popped first. In the new stack the two 9s would be the first values popped from the stack. You may use a single queue as auxiliary storage to solve this problem.

```
public void stutter(Stack<Integer> s) {
   Queue<Integer> q = new LinkedList<Integer>();
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   while (!q.isEmpty()) {
      s.push(q.remove());
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   while(!q.isEmpty()) {
      int n = q.remove();
      s.push(n);
      s.push(n);
   }
}
```

Exercise 14.3: copyStack

Write a method copyStack that takes a stack of integers as a parameter and returns a copy of the original stack (i.e., a new stack with the same values as the original, stored in the same order as the original). Your method should create the new stack and fill it up with the same values that are stored in the original stack. It is not acceptable to return the same stack passed to the method; you must create, fill, and return a new stack.

You will be removing values from the original stack to make the copy, but you have to be sure to put them back into the original stack in the same order before you are done. In other words, when your method is done executing, the original stack must be restored to its original state and you will return the new independent stack that is in the same state. You may use one queue as auxiliary storage.

```
public Stack<Integer> copyStack(Stack<Integer> s) {
   Stack<Integer> s2 = new Stack<Integer>();
   Queue<Integer> q = new LinkedList<Integer>();
   while (!s.isEmpty()) {
      s2.push(s.pop());
   }
   while(!s2.isEmpty()) {
        q.add(s2.pop());
    }
    while (!q.isEmpty()) {
        int n = q.remove();
        s.push(n);
        s2.push(n);
    }
    return s2;
}
```

Exercise 14.4: collapse

Write a method collapse that takes a stack of integers as a parameter and that collapses it by replacing each successive pair of integers with the sum of the pair. For example, suppose a stack stores these values:

bottom [7, 2, 8, 9, 4, 13, 7, 1, 9, 10] top

The first pair should be collapsed into 9 (7 + 2), the second pair should be collapsed into 17 (8 + 9), the third pair should be collapsed into 17 (4 + 13) and so on to yield:

bottom [9, 17, 17, 8, 19] top

If the stack stores an odd number of elements, the final element is not collapsed. For example, the stack:

bottom [1, 2, 3, 4, 5] top

Would collapse into:

bottom [3, 7, 5] top

With the 5 at the top of the stack unchanged. You may use one queue as auxiliary storage.

```
public static void collapse(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<Integer>();
    while (!s.isEmpty()) {
        q.add(s.pop());
    }
    while (!q.isEmpty()) {
        s.push(q.remove());
    }
    while (!s.isEmpty()) {
        q.add(s.pop());
    }
    while (q.size() > 1) {
        s.push(q.remove() + q.remove());
    }
    if (!q.isEmpty()) {
        s.push(q.remove());
    }
}
```

Exercise 14.5: euqlas

Write a method equals that takes as parameters two stacks of integers and returns true if the two stacks are equal and that returns false otherwise. To be considered equal, the two stacks would have to store the same sequence of integer values in the same order. Your method is to examine the two stacks but must return them to their original state before terminating. You may use one stack as auxiliary storage.

```
public boolean equals(Stack<Integer> s1, Stack<Integer> s2) {
    Stack<Integer> s3 = new Stack<Integer>();
    boolean same = true;
    while (same && !s1.isEmpty() && !s2.isEmpty()) {
        int num1 = s1.pop();
        int num2 = s2.pop();
        if (num1 != num2) {
            same = false;
        }
        s3.push(num1);
        s3.push(num2);
    }
    same = same && s1.isEmpty() && s2.isEmpty();
    while (!s3.isEmpty()) {
        s2.push(s3.pop());
        s1.push(s3.pop());
    }
    return same;
}
```

Exercise 14.6: rearrange

Write a method rearrange that takes a queue of integers as a parameter and rearranges the order of the values so that all of the even values appear before the odd values and that otherwise preserves the original order of the list. For example, suppose a queue called q stores this sequence of values:

front [3, 5, 4, 17, 6, 83, 1, 84, 16, 37] back

Then the call of rearrange(q); should rearrange the queue to store the following sequence of values:

front [4, 6, 84, 16, 3, 5, 17, 83, 1, 37] back

Notice that all of the evens appear at the front of the queue followed by the odds and that the order of the evens is the same as in the original list and the order of the odds is the same as in the original list. You may use one stack as auxiliary storage.

```
public void rearrange(Queue<Integer> q) {
   Stack<Integer> s = new Stack<Integer>();
   int oldSize = q.size();
   for (int i = 0; i < oldSize; i++) {
      int n = q.remove();
      if (n % 2 == 0) {
         q.add(n);
      } else {
         s.push(n);
      }
   }
   int evenCount = q.size();
   while (!s.isEmpty()){
      q.add(s.pop());
   }
   for (int i = 0; i < evenCount; i++) {
      q.add(q.remove());
   }
   for (int i = 0; i < oldSize - evenCount; i++) {
      s.push(q.remove());
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
}
```

Exercise 14.7: reverseHalf

Write a method reverseHalf that reverses the order of half of the elements of a Queue of integers. Your method should reverse the order of all the elements in odd-numbered positions (position 1, 3, 5, etc.) assuming that the first value in the queue has position 0. For example, if the queue originally stores this sequence of numbers when the method is called:

```
index: 0  1  2  3  4  5   6   7
front [1, 8, 7, 2, 9, 18, 12, 0] back
```

- it should store the following values after the method finishes executing:

```
index: 0  1  2  3   4  5  6   7
front [1, 0, 7, 18, 9, 2, 12, 8] back
```

Notice that numbers in even positions (positions 0, 2, 4, 6) have not moved. That sub-sequence of numbers is still: (1, 7, 9, 12). But notice that the numbers in odd positions (positions 1, 3, 5, 7) are now in reverse order relative to the original. In other words, the original sub-sequence: (8, 2, 18, 0) - has become: (0, 18, 2, 8). You may use a single stack as auxiliary storage.

```java
public void reverseHalf(Queue<Integer> q) {
   Stack<Integer> s = new Stack<Integer>();

   int oldSize = q.size();

   for(int i = 0; i < oldSize; i++) {
      if(i % 2 == 0) {
         q.add(q.remove());
      } else {
         s.push(q.remove());
      }
   }

   for(int i = 0; i < oldSize; i++) {
      if(i % 2 == 0) {
         q.add(q.remove());
      } else {
         q.add(s.pop());
      }
   }
}
```

Exercise 14.8: isPalindrome

Write a method isPalindrome that takes a queue of integers as a parameter and returns true if the numbers in the queue represent a palindrome (and false otherwise). A sequence of numbers is considered a palindrome if it is the same in reverse order. For example, suppose a queue called q stores these values:

```
front [3, 8, 17, 9, 17, 8, 3] back
```

Then the call of isPalindrome(q); should return true because this sequence is the same in reverse order. If the queue had instead stored these values:

```
front [3, 8, 17, 9, 4, 17, 8, 3] back
```

The call on isPalindrome would instead return false because this sequence is not the same in reverse order (the 9 and 4 in the middle don't match). The empty queue should be considered a palindrome. You may not make any assumptions about how many elements are in the queue and your method must restore the queue so that it stores the same sequence of values after the call as it did before. You may use one stack as auxiliary storage.

```
public boolean isPalindrome(Queue<Integer> q) {
   Stack<Integer> s = new Stack<Integer>();
   for (int i = 0; i < q.size(); i++) {
      int n = q.remove();
      q.add(n);
      s.push(n);
   }
   boolean ok = true;
   for (int i = 0; i < q.size(); i++) {
      int n1 = q.remove();
      int n2 = s.pop();
      if (n1 != n2) {
         ok = false;
      }
      q.add(n1);
   }
   return ok;
}
```

Exercise 14.9: switchPairs

Write a method switchPairs that takes a stack of integers as a parameter and that switches successive pairs of numbers starting at the bottom of the stack. For example, if the stack initially stores these values:

bottom [3, 8, 17, 9, 99, 9, 17, 8, 3, 1, 2, 3, 4, 14] top

Your method should switch the first pair (3, 8), the second pair (17, 9), the third pair (99, 9), and so on, yielding this sequence:

bottom [8, 3, 9, 17, 9, 99, 8, 17, 1, 3, 3, 2, 14, 4] top

If there are an odd number of values in the stack, the value at the top of the stack is not moved. For example, if the original stack had stored:

bottom [3, 8, 17, 9, 99, 9, 17, 8, 3, 1, 2, 3, 4, 14, 42] top

It would again switch pairs of values, but the value at the top of the stack (42) would not be moved, yielding this sequence:

bottom [8, 3, 9, 17, 9, 99, 8, 17, 1, 3, 3, 2, 14, 4, 42] top

Do not make assumptions about how many elements are in the stack. Use one queue as auxiliary storage.

```
public void switchPairs(Stack<Integer> s) {
   Queue<Integer> q = new LinkedList<Integer>();
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   while (!q.isEmpty()) {
      s.push(q.remove());
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   while (q.size() > 1) {
      int n1 = q.remove();
      int n2 = q.remove();
      s.push(n2);
      s.push(n1);
   }
   if (!q.isEmpty()) {
      s.push(q.remove());
   }
}
```

Exercise 14.10: isCOnsecutive
Write a method isConsecutive that takes a stack of integers as a parameter and that returns whether or not the stack contains a sequence of consecutive integers starting from the bottom of the stack (returning true if it does, returning false if it does not). Consecutive integers are integers that come one after the other, as in 5, 6, 7, 8, 9, etc. So if a stack s stores the following values:

bottom [3, 4, 5, 6, 7, 8, 9, 10] top

Then the call of isConsecutive(s) should return true. If the stack had instead contained this set of values:

bottom [3, 4, 5, 6, 7, 8, 9, 10, 12] top

Then the call should return false because the numbers 10 and 12 are not consecutive. Notice that we look at the numbers starting at the bottom of the stack. The following sequence of values would be consecutive except for the fact that it appears in reverse order, so the method would return false:

bottom [3, 2, 1] top

Your method must restore the stack so that it stores the same sequence of values after the call as it did before. Any stack with fewer than two values should be considered to be a list of consecutive integers. You may use one queue as auxiliary storage to solve this problem.

```
public boolean isConsecutive(Stack<Integer> s) {
   if (s.size() <= 1) {
      return true;
   } else {
      Queue<Integer> q = new LinkedList<Integer>();
      int prev = s.pop();
      q.add(prev);
      boolean ok = true;
      while (!s.isEmpty()) {
         int next = s.pop();
         if (prev - next != 1) {
            ok = false;
         }
         q.add(next);
         prev = next;
      }
      while (!q.isEmpty()) {
         s.push(q.remove());
      }
      while (!s.isEmpty()) {
         q.add(s.pop());
      }
      while (!q.isEmpty()) {
         s.push(q.remove());
      }
      return ok;
   }
}
```

Exercise 14.11: reorder

Write a method reorder that takes a queue of integers as a parameter and that puts the integers into sorted (nondecreasing) order assuming that the queue is already sorted by absolute value. For example, suppose that a variable called q stores the following sequence of values:

front [1, 2, -2, 4, -5, 8, -8, 12, -15, 23] back

Notice that the values appear in sorted order if you ignore the sign of the numbers. The call of reorder(q); should reorder the values so that the queue stores this sequence of values:

front [-15, -8, -5, -2, 1, 2, 4, 8, 12, 23] back

Notice that the values now appear in sorted order taking into account the sign of the numbers. You may use one stack as auxiliary storage to solve this problem.

```
public void reorder(Queue<Integer> q) {
   Stack<Integer> s = new Stack<Integer>();
   int oldSize = q.size();
   for (int i = 0; i < oldSize; i++) {
      int n = q.remove();
      if (n < 0) {
         s.push(n);
      } else {
         q.add(n);
      }
   }
   int newSize = q.size();
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   for (int i = 0; i < newSize; i++) {
      q.add(q.remove());
   }
}
```

Exercise 14.12: shift

Write a method shift that takes a stack of integers and an integer *n* as parameters and that shifts *n* values from the bottom of the stack to the top of the stack. For example, if a variable called s stores the following sequence of values:

bottom [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] top

If we make the call shift(s, 6); the method shifts the six values at the bottom of the stack to the top of the stack and leaves the other values in the same order producing:

bottom [7, 8, 9, 10, 6, 5, 4, 3, 2, 1] top

Notice that the value that was at the bottom of the stack is now at the top, the value that was second from the bottom is now second from the top, the value that was third from the bottom is now third from the top, and so on, and that the four values not involved in the shift are now at the bottom of the stack in their original order. If s had stored these values instead:

bottom [7, 23, -7, 0, 22, -8, 4, 5] top

If we make the following call: shift(s, 3); then s should store these values after the call:

bottom [0, 22, -8, 4, 5, -7, 23, 7] top

You are to use one queue as auxiliary storage to solve this problem. You may assume that the parameter *n* is >= 0 and not larger than the number of elements in the stack.

```
public void shift(Stack<Integer> s, int n) {
   Queue<Integer> q = new LinkedList<Integer>();
   int otherSize = s.size() - n;
   for (int i = 0; i < otherSize; i++) {
      q.add(s.pop());
   }
   while (!q.isEmpty()) {
      s.push(q.remove());
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   while (!q.isEmpty()) {
      s.push(q.remove());
   }
}
```

Exercise 14.13: expunge

Write a method expunge that accepts a stack of integers as a parameter and makes sure that the stack's elements are in non-decreasing order from top to bottom, by removing from the stack any element that is smaller than any element(s) on top of it. For example, suppose a variable s stores the following elements:

bottom [4, 20, 15, 15, 8, 5, 7, 12, 3, 10, 5, 0] top

The element values 3, 7, 5, 8, and 4 should be removed because each has an element above it with a larger value. So the call of expunge(s); should change the stack to store the following elements in this order:

bottom [20, 15, 15, 12, 10, 5, 0] top

Notice that now the elements are in non-decreasing order from top to bottom. If the stack is empty or has just one element, nothing changes. You may assume that the stack passed is not null.

(Hint: An element *e* that should be removed is one that is smaller than some element above *e*. But since the elements above *e* are in sorted order, you may not need to examine all elements above e in order to know whether to remove *e*.)

Obey the following restrictions in your solution:

- You may use one queue or stack (but not both) as auxiliary storage. You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
- Use stacks/queues in stack/queue-like ways only. Do not call index-based methods such as get, search, or set (or for-each) on a stack/queue. You may call only add, remove, push, pop, peek, isEmpty, and size.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
public static void s2q(Stack s, Queue q) { ... }
public static void q2s(Queue q, Stack s) { ... }
```

```
// using a temporary stack
public static void expunge(Stack<Integer> s) {
   if (!s.isEmpty()) {
      // copy sorted contents into temp stack s2
      Stack<Integer> s2 = new Stack<Integer>();
      int prev;
      while (!s.isEmpty()) {
         prev = s.pop();
         while (!s.isEmpty() && s.peek() < prev) {
            s.pop();
         }
         s2.push(prev);
      }
```

```
        // transfer s2 back into s
        while (!s2.isEmpty()) {
            s.push(s2.pop());
        }
    }
}
```

Exercise 14.14: reverseFirstK

Write a method reverseFirstK that accepts an integer *k* and a queue of integers as parameters and reverses the order of the first *k* elements of the queue, leaving the other elements in the same relative order. For example, suppose a variable q stores the following elements:

front [10, 20, 30, 40, 50, 60, 70, 80, 90] back

The call of reverseFirstK(4, q); should change the queue to store the following elements in this order:

front [40, 30, 20, 10, 50, 60, 70, 80, 90] back

If *k* is 0 or negative, no change should be made to the queue. If the queue passed is null or does not contain at least *k* elements, your method should throw an IllegalArgumentException.

For full credit, obey the following restrictions in your solution. A solution that disobeys them can get partial credit.

- You may use one queue or stack (but not both) as auxiliary storage.
- You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
- Use the Queue interface and Stack/LinkedList classes discussed in lecture.
- Use stacks/queues in stack/queue-like ways only. Do not call index-based methods such as get, search, or set (or use a for-each loop) on a stack/queue. You may call only add, remove, push, pop, peek, isEmpty, and size.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
public static void s2q(Stack s, Queue q) { ... }
public static void q2s(Queue q, Stack s) { ... }
```

```
public static void reverseFirstK(int k, Queue<Integer> q) {
    if (q == null || k > q.size()) {
        throw new IllegalArgumentException();
    } else if (k > 0) {
        Stack<Integer> s = new Stack<Integer>();   // first k elements -> S
        for (int i = 0; i < k; i++) {
           s.push(q.remove());
        }

        while (!s.isEmpty()) {                            // s2q(s, q);
           q.add(s.pop());
        }

        for (int i = 0; i < q.size() - k; i++) {   // wrap around rest of elements so
           q.add(q.remove());                      // k reversed ones appear at front
        }
    }
}
```

Exercise 14.15: isSorted

Write a method isSorted that accepts a stack of integers as a parameter and returns true if the elements in the stack occur in ascending (non-decreasing) order from top to bottom, and false otherwise. That is, the smallest element should be on top, growing larger toward the bottom. For example, passing the following stack to your method should cause it to return true:

bottom [20, 20, 17, 11, 8, 8, 3, 2] top

The following stack is not sorted (the 15 is out of place), so passing it to your method should return a result of false:

bottom [18, 12, 15, 6, 1] top

An empty or one-element stack is considered to be sorted. When your method returns, the stack should be in the same state as when it was passed in. In other words, if your method modifies the stack, you must restore it before returning.

Obey the following restrictions in your solution:

- You may use one queue or stack (but not both) as auxiliary storage.
- You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
- Use the Queue interface and Stack/LinkedList classes discussed in the textbook.
- Use stacks/queues in stack/queue-like ways only. Do not call index-based methods such as get, search, or set (or use a for-each loop or iterator) on a stack/queue. You may call only add, remove, push, pop, peek, isEmpty, and size.
- Your solution should run in O(N) time, where N is the number of elements of the stack.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
public static void s2q(Stack s, Queue q) { ... }
public static void q2s(Queue q, Stack s) { ... }
```

```
public static boolean isSorted(Stack<Integer> s) {
   if (s.size() < 2) {
      return true;
   }

   boolean sorted = true;
   int prev = s.pop();
   Stack<Integer> backup = new Stack<Integer>();
   backup.push(prev);
   while (!s.isEmpty()) {
      int curr = s.pop();
      backup.push(curr);
      if (prev > curr) {
         sorted = false;
      }
      prev = curr;
   }

   while (!backup.isEmpty()) {   // restore s
      s.push(backup.pop());
   }

   return sorted;
}
```

Exercise 14.16: mirror

Write a method mirror that accepts a stack of integers as a parameter and replaces the stack contents with itself plus a mirrored version of itself (the same elements in the opposite order). For example, suppose a variable s stores the following elements:

bottom [10, 50, 19, 54, 30, 67] top

After a call of mirror(s);, the stack would store the following elements (underlined for emphasis):

bottom [10, 50, 19, 54, 30, 67, 67, 30, 54, 19, 50, 10] top

Note that the mirrored version is added on to the top of what was originally in the stack. The bottom half of the stack contains the original numbers in the same order. If your method is passed an empty stack, the result should be an empty stack. If your method is passed a null stack, your method should throw an IllegalArgumentException.

You may use one stack or one queue (but not both) as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You may not use recursion to solve this problem. For full credit your code must run in O(n) time where n is the number of elements of the original stack. Use the Queue interface and Stack/LinkedList classes from lecture.

```
public static void mirror(Stack<Integer> s) {
   if (s == null)
      throw new IllegalArgumentException();
   Queue<Integer> q = new LinkedList<Integer>();

   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   for (int i = 0; i < q.size(); i++) {
      int n = q.remove();
      q.add(n);
      s.push(n);
   }
   int size = q.size();
   while (!s.isEmpty()) {
      q.add(s.pop());
   }
   for(int i = 0; i < size; i++) {
      q.add(q.remove());
   }
   while (!q.isEmpty()) {
      s.push(q.remove());
   }
}
```

Exercise 14.17: compressDuplicates

Write a method compressDuplicates that accepts a stack of integers as a parameter and that replaces each sequence of duplicates with a pair of values: a count of the number of duplicates, followed by the actual duplicated number. For example, suppose a variable called s stores the following sequence of values:

bottom [2, 2, 2, 2, 2, -5, -5, 3, 3, 3, 3, 4, 4, 1, 0, 17, 17] top

If we make the call of compressDuplicates(s);, after the call s should store the following values:

bottom [5, 2, 2, -5, 4, 3, 2, 4, 1, 1, 1, 0, 2, 17] top

This new stack indicates that the original had 5 occurrences of 2 at the bottom of the stack followed by 2 occurrences of -5 followed by 4 occurrences of 3, and so on. This process works best when there are many duplicates in a row. For example, if the stack instead had stored:

bottom [10, 20, 10, 20, 20, 10] top

Then the resulting stack after the call ends up being longer than the original:

bottom [1, 10, 1, 20, 1, 10, 2, 20, 1, 10] top

If the stack is empty, your method should not change it. You may use one queue as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You may not use recursion to solve this problem. For full credit your code must run in O(*n*) time where *n* is the number of elements of the original stack.

```
public void compressDuplicates(Stack<Integer> s) {
   Queue<Integer> q = new LinkedList<Integer>();
   while (!s.isEmpty()) {
      q.add(s.pop());      // s -> q
   }
   while (!q.isEmpty()) {
      s.push(q.remove());   // q -> s, to reverse the stack order
   }
   while (!s.isEmpty()) {     // s -> q
      q.add(s.pop());
   }
   if (!q.isEmpty()) {      // q -> s, replacing dupes with (count, val)
      int last = q.remove();
      int count = 1;
      while (!q.isEmpty()) {
         int next = q.remove();
         if (next == last) {
            count++;
         } else {
            s.push(count);
            s.push(last);
            count = 1;
            last = next;
            }
        }
        s.push(count);
        s.push(last);
    }
}
```

Exercise 14.18: mirrorHalves

Write a method mirrorHalves that accepts a queue of integers as a parameter and replaces each half of that queue with itself plus a mirrored version of itself (the same elements in the opposite order). For example, suppose a queue variable q stores the following elements:

front [10, 50, 19, 54, 30, 67] back

After a call of mirrorHalves(q);, the queue would store the following elements:

front [10, 50, 19, 19, 50, 10, 54, 30, 67, 67, 30, 54] back

If your method is passed an empty queue, the result should be an empty queue. If your method is passed a null queue or one whose size is not even, your method should throw an IllegalArgumentException.

You may use one stack or one queue (but not both) as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You may not use recursion to solve this problem. For full credit your code must run in O(*n*) time where *n* is the number of elements of the original queue. Use the Queue interface and Stack/LinkedList classes from lecture.

```
public static void mirrorHalves(Queue<Integer> q) {
    if (q == null || q.size() % 2 != 0) {
        throw new IllegalArgumentException();
    }

    Stack<Integer> s = new Stack<Integer>();
    int size = q.size();

    for (int i = 0; i < size / 2; i++) {
        int element = q.remove();
        s.push(element);
        q.add(element);
    }
    while (!s.isEmpty()) {
        q.add(s.pop());
    }

    for (int i = 0; i < size / 2; i++) {
        int element = q.remove();
        s.push(element);
        q.add(element);
    }
    while (!s.isEmpty()) {
        q.add(s.pop());
    }
}
```

Exercise 14.19: removeMin

Write a method removeMin that accepts a Stack of integers as a parameter and removes and returns the smallest value from the stack. For example if a variable *s* stores these values:

bottom [2, 8, 3, 19, 7, 3, 2, 3, 2, 7, 12, -8, 4] top

and we make the following call:

```
int n = removeMin(s);
```

The method removes and returns -8, so *n* will store -8 after the call and *s* will store the following values:

bottom [2, 8, 3, 19, 7, 3, 2, 3, 2, 7, 12, 4] top

If the minimum value appears more than once, all occurrences of it should be removed. For example, given the stack above, if we again call removeMin(s), it would return 2 and leave the stack as follows:

bottom [8, 3, 19, 7, 3, 3, 7, 12, 4] top

You may use one queue as auxiliary storage. You may not use any other structures to solve this problem, although you can have as many primitive variables as you like. You may not solve the problem recursively.

```
public static int removeMin(Stack<Integer> s) {
    Queue<Integer> q = new LinkedList<Integer>();

    int min = s.pop();
    q.add(min);
    while(!s.isEmpty()) {
        int next = s.pop();
        if(next < min) {
            min = next;
        }
        q.add(next);
    }

    while(!q.isEmpty()) {
        int next = q.remove();
        if(next != min) {
            s.push(next);
        }
    }

    s2q(s, q);
    q2s(q, s);

    return min;
}
```

Exercise 14.20: interleave

Write a method interleave that accepts a queue of integers as a parameter and rearranges the elements by alternating the elements from the first half of the queue with those from the second half of the queue. For example, suppose a variable q stores the following sequence of values:

front [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] back

If we make the call of interleave(q);, the queue should store the following values after the call:

front [1, 6, 2, 7, 3, 8, 4, 9, 5, 10] back

To understand the result, consider the two halves of this list. The first half is [1, 2, 3, 4, 5] and the second half is [6, 7, 8, 9, 10]. These are combined in an alternating fashion to form a sequence of interleave pairs: the first values from each half (1 and 6), then the second values from each half (2 and 7), then the third values from each half (3 and 8), and so on. In each pair, the value from the first half appears before the value from the second half. The previous example uses sequential integers to make the interleaving more obvious, but the same process can be applied to any sequence of even length. For example, if q had instead stored these values:

front [2, 8, -5, 19, 7, 3, 24, 42] back

Then the method would have rearranged the list to become:

front [2, 7, 8, 3, -5, 24, 19, 42] back

Your method should throw an IllegalArgumentException if the queue does not have even size. You may use one stack as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You may not use recursion to solve this problem. For full credit, your solution must run in O(*n*) time, where *n* represents the size of the queue.

```
public static void interleave(Queue<Integer> q) {
   if (q.size() % 2 != 0) {
      throw new IllegalArgumentException();
   }
   Stack<Integer> s = new Stack<Integer>();
   int halfSize = q.size() / 2;
   for (int i = 0; i < halfSize; i++) {
      s.push(q.remove());
   }
   while (!s.isEmpty()) {   // s2q(s, q);
      q.add(s.pop());
   }
   for (int i = 0; i < halfSize; i++) {
      q.add(q.remove());
   }
   for (int i = 0; i < halfSize; i++) {
      s.push(q.remove());
   }
   while (!s.isEmpty()) {
      q.add(s.pop());
      q.add(q.remove());
   }
}
```

Chapter 15: Implementing a Collection Class

Exercise 15.5: runnignTotal
Write a method runningTotal that returns a new ArrayIntList that contains a running total of the original list. In other words, the i th value in the new list should store the sum of elements 0 through *i* of the original list. For example, if a variable list stores the following sequence of values:

[2, 3, 5, 4, 7, 15, 20, 7]

and the following call is made:

ArrayIntList list2 = list.runningTotal();

Then the variable list2 should store the following sequence of values:

[2, 5, 10, 14, 21, 36, 56, 63]

The original list should not be changed by the method. If the original list is empty, the result should be an empty list. The new list should have the same capacity as the original. Remember that there is a constructor for ArrayIntList that takes a capacity as a parameter:

```
// Constructs an empty list with the given capacity.
// Precondition: capacity >= 0
public ArrayIntList(int capacity)
```

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   // your code goes here
}
```

```
public ArrayIntList runningTotal() {
   ArrayIntList result = new ArrayIntList(elementData.length);
   if (size > 0) {
      result.add(elementData[0]);
      for (int i = 1; i < size; i++) {
         result.add(result.get(i - 1) + elementData[i]);
      }
   }
   return result;
}
```

Exercise 15.7: isPairwiseSorted

Write a method isPairwiseSorted that returns whether or not a list of integers is pairwise sorted (true if it is, false otherwise). A list is considered pairwise sorted if each successive pair of numbers is in sorted (non-decreasing) order. For example, if a variable called list stores the following sequence of values:

[3, 8, 2, 5, 19, 24, -3, 0, 4, 4, 8, 205, 42]

Then the call of list.isPairwiseSorted() should return true because the successive pairs of this list are all sorted: (3, 8), (2, 5), (19, 24), (-3, 0), (4, 4), (8, 205). Notice that the extra value 42 at the end had no effect on the result because it is not part of a pair. If the list had instead stored the following:

[1, 9, 3, 17, 4, 28, -5, -3, 0, 42, 308, 409, 19, 17, 2, 4]

Then the method should return false because the pair (19, 17) is not in sorted order. If a list is so short that it has no pairs, then it is considered to be pairwise sorted. If a list is of odd length, the final element should be ignored since it has no pair. In other words, if the rest of the list is pairwise sorted until that last unpaired element, your method should return true.

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   // your code goes here
}
```

```
public boolean isPairwiseSorted() {
   for (int i = 0; i < size - 1; i += 2) {
      if (elementData[i] > elementData[i + 1]) {
         return false;
      }
   }
   return true;
}
```

Exercise 15.10: longestSortedSequence

Write a method longestSortedSequence that returns the length of the longest sorted sequence within a list of integers. For example, if a variable called list stores the following sequence of values:

[1, 3, 5, 2, 9, 7, -3, 0, 42, 308, 17]

then the call: list.longestSortedSequence() would return the value 4 because it is the length of the longest sorted sequence within this list (the sequence -3, 0, 42, 308). If the list is empty, your method should return 0. Notice that for a non-empty list the method will always return a value of at least 1 because any individual element constitutes a sorted sequence.

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   // your code goes here
}

public int longestSortedSequence() {
   if (size == 0) {
      return 0;
   }
   int max = 1;
   int current = 1;
   for (int i = 1; i < size; i++) {
      if (elementData[i] >= elementData[i - 1]) {
         current++;
         if (current > max) {
            max = current;
         }
      } else {
         current = 1;
      }
   }
   return max;
}
```

Exercise 15.12: removeFront

Write a method removeFront that takes an integer *n* as a parameter and that removes the first *n* values from a list of integers. For example, if a variable called list stores this sequence of values:

[8, 17, 9, 24, 42, 3, 8]

and the following call is made:

list.removeFront(4);

It should store the following after the call:

[42, 3, 8]

Notice that the first four values in the list have been removed and the other values appear in the same order as in the original list. You may assume that the parameter value passed is between 0 and the size of the list inclusive.

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   // your code goes here
}

public void removeFront(int n) {
   for (int i = n; i < size; i++) {
      elementData[i - n] = elementData[i];
   }
   size -= n;
}
```

Exercise 15.13: removeAll

Write a method removeAll that takes an integer value as a parameter and that removes all occurrences of the given value from the list. You can assume that a method called remove exists that takes an index as a parameter and that removes the value at the given index.

For example, if the variable named list stores the following values:

[14, 5, -1, 7, 14, 7, 7, 29, 3, 7]

After the call of list.removeAll(7); the list would store the following values:

[14, 5, -1, 14, 29, 3]

Assume you are adding to the ArrayIntList class with following fields & methods:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   public void remove(int index) { ... }

   // your code goes here
}
```

```
public void removeAll(int value) {
   int i = 0;
   while (i < size) {
      if (elementData[i] == value) {
         remove(i);
      } else {
         i++;
      }
   }
}
```

Exercise 15.14: printInversions

Write a method printInversions that lists all inversions in a list of integers. An inversion is defined as a pair of numbers where the first appears before the second in the list, but the first is greater than the second. Thus, for a sorted list such as [1, 2, 3, 4] there are no inversions at all, and printInversions would produce no output. Suppose that a variable called list stores a list in reverse order, as in [4, 3, 2, 1]. Then the call: list.printInversions(); would print many inversions:

```
(4, 3)
(4, 2)
(4, 1)
(3, 2)
(3, 1)
(2, 1)
```

You must reproduce this format exactly and output your inversions in the same order. Notice that any given number (e.g., 4 in the list above) can produce several different inversions, because there might be several numbers after it that are less than it (all of 1, 2, and 3 in the example). You may assume that the list has no duplicates.

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   // your code goes here
}

public void printInversions() {
   for (int i = 0; i < size - 1; i++) {
      for (int j = i + 1; j < size; j++) {
         if (elementData[i] > elementData[j]) {
            System.out.println("(" + elementData[i] + ", "
                           + elementData[j] + ")");
         }
      }
   }
}
```

Exercise 15.15: mirror

Write a method mirror that doubles the size of a list of integers by appending the mirror image of the original sequence to the end of the list. The mirror image is the same sequence of values in reverse order. For example, if a variable called list stores this sequence of values:

[1, 3, 2, 7]

and we make the following call:

list.mirror();

then it should store the following values after the call:

[1, 3, 2, 7, 7, 2, 3, 1]

Notice that it has been doubled in size by having the original sequence appearing in reverse order at the end of the list. You may not make assumptions about how many elements are in the list. Because adding these elements might overrun the capacity of the underlying array, you may need to call ensureCapacity to enlarge this array.

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   public void add(int value) { ... }
   public void add(int index, int value) { ... }
   public void ensureCapacity(int capacity) { ... }
   ...

   // your code goes here
}
```

```
public void mirror() {
   ensureCapacity(size * 2);
   int last = 2 * size - 1;
   for (int i = 0; i < size; i++) {
      elementData[last - i] = elementData[i];
   }
   size = size * 2;
}
```

Exercise 15.17: stretch

Write a method stretch that takes an integer *n* as a parameter and that increases a list of integers by a factor of *n* by replacing each integer in the original list with *n* copies of that integer. For example, if a variable called list stores this sequence of values:

[18, 7, 4, 24, 11]

And we make the following call:

list.stretch(3);

The list should store the following values after the method is called:

[18, 18, 18, 7, 7, 7, 4, 4, 4, 24, 24, 24, 11, 11, 11]

If the value of *n* is less than or equal to 0, the list should be empty after the call.

Because adding elements might overrun the capacity of the underlying array, you may need to call ensureCapacity to enlarge this array.

Assume you are adding to the ArrayIntList class with following fields:

```
public class ArrayIntList {
   private int[] elementData;
   private int size;

   public void add(int value) { ... }
   public void add(int index, int value) { ... }
   public void ensureCapacity(int capacity) { ... }
   ...

   // your code goes here
}
```

```
public void stretch(int n) {
   if (n > 0) {
      ensureCapacity(n * size);
      for (int i = size - 1; i >= 0; i--) {
         for (int j = 0; j < n; j++) {
            elementData[i * n + j] = elementData[i];
         }
      }
      size *= n;
   } else {
      size = 0;
   }
}
```

Chapter 16: Linked Lists

Exercise 16.1: set

Write a method set that accepts an index and a value and sets the list's element at that index to have the given value. You may assume that the index is between 0 (inclusive) and the size of the list (exclusive).

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void set(int index, int value) {
   ListNode current = front;
   for (int i = 0; i < index; i++) {
      current = current.next;
   }
   current.data = value;
}
```

Exercise 16.2: min

Write a method min that returns the minimum value in a list of integers. If the list is empty, it should throw a NoSuchElementException.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public int min() {
   if (front == null) {
      throw new NoSuchElementException("list is empty");
   } else {
      int min = front.data;
      ListNode current = front.next;
      while (current != null) {
         if (current.data < min) {
            min = current.data;
         }
         current = current.next;
```

```
        }
        return min;
    }
}
```

Exercise 16.3: isSorted

Write a method isSorted that returns true if the list is in sorted (nondecreasing) order and returns false otherwise. An empty list is considered to be sorted.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
    private ListNode front;   // null for an empty list
    ...
}

public boolean isSorted() {
    if (front != null) {
        ListNode current = front;
        while (current.next != null) {
            if (current.data > current.next.data) {
                return false;
            }
            current = current.next;
        }
    }
    return true;
}
```

Exercise 16.4: lastIndexOf

Write a method lastIndexOf that accepts an integer value as a parameter and that returns the index in the list of the last occurrence of that value, or -1 if the value is not found in the list. For example, if a variable list stores the following sequence of values, then the call of list.lastIndexOf(18) should return 6 because that is the index of the last occurrence of 18:

[1, 18, 2, 7, 18, 39, 18, 40]

If the call had instead been list.lastIndexOf(3), the method would return -1 because 3 does not appear in the list. You may not call any other methods of the class to solve this problem.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public int lastIndexOf(int n) {
   int result = -1;
   int index = 0;
   ListNode current = this.front;
   while (current != null) {
      if (current.data == n) {
         result = index;
      }
      index++;
      current = current.next;
   }
   return result;
}
```

Exercise 16.5: countDuplicates

Write a method countDuplicates that returns the number of duplicates in a sorted list. The list will be in sorted order, so all of the duplicates will be grouped together. For example, if a variable list stores the sequence of values below, the call of list.countDuplicates() should return 7 because there are 2 duplicates of 1, 1 duplicate of 3, 1 duplicate of 15, 2 duplicates of 23 and 1 duplicate of 40:

[1, 1, 1, 3, 3, 6, 9, 15, 15, 23, 23, 23, 40, 40]

Remember that you may assume that the list is in sorted order, so any duplicates would occur consecutively.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public int countDuplicates() {
   int count = 0;
   if (front != null) {
      ListNode current = front;
      while (current.next != null) {
         if (current.data == current.next.data) {
            count++;
            }
            current = current.next;
        }
    }
    return count;
}
```

Exercise 16.6: hasTwoConsecutive

Write a method hasTwoConsecutive that returns whether or not a list of integers has two adjacent numbers that are consecutive integers (true if such a pair exists and false otherwise). For example, if a variable list stores the following sequence of values, then the call list.hasTwoConsecutive() should return true because the list contains the adjacent numbers (7, 8) which are a pair of consecutive numbers:

[1, 18, 2, 7, 8, 39, 18, 40]

If the list had stored the following sequence of values, then the method should return false:

[1, 18, 17, 2, 7, 39, 18, 40, 8]

This sequence contains some pairs of numbers that could represent consecutive integers (e.g., 1 and 2, 7 and 8, 39 and 40), but those pairs of numbers are not adjacent in the sequence. The list also has a pair of adjacent numbers (18, 17) that are not in the right order to be considered consecutive. You may not make any assumptions about how many elements are in the list.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
    private ListNode front;   // null for an empty list
    ...
}

public boolean hasTwoConsecutive() {
    if (front == null || front.next == null) {
        return false;
    }
    ListNode current = front;
    while (current.next != null) {
        if (current.data + 1 == current.next.data) {
            return true;
        }
        current = current.next;
    }
```

```
   return false;
}
```

Exercise 16.7: deleteBack

Write a method deleteBack that deletes the last value (the value at the back of the list) and returns the deleted value. If the list is empty, your method should throw a NoSuchElementException.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}
```

```
public int deleteBack() {
   if (front == null) {
      throw new NoSuchElementException("empty list");
   }
   int result = 0;
   if (front.next == null) {
      result = front.data;
      front = null;
   } else {
      ListNode current = front;
      while (current.next.next != null) {
         current = current.next;
      }
      result = current.next.data;
      current.next = null;
   }
   return result;
}
```

Exercise 16.8: switchPairs

Write a method switchPairs that switches the order of elements in a linked list of integers in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. For example, if the list initially stores these values:

[3, 7, 4, 9, 8, 12]

Your method should switch the first pair (3, 7), the second pair (4, 9), the third pair (8, 12), etc. to yield this list:

[7, 3, 9, 4, 12, 8]

If there are an odd number of values, the final element is not moved. For example, if the list had been:

[3, 7, 4, 9, 8, 12, 2]

It would again switch pairs of values, but the final value (2) would not be moved, yielding this list:

[7, 3, 9, 4, 12, 8, 2]

Assume that we are adding this method to the LinkedIntList class as shown below. You may not call any other methods of the class to solve this problem, you may not construct any new nodes, and you may not use any auxiliary data structures to solve this problem (such as an array, ArrayList, Queue, String, etc.). You also may not change any data fields of the nodes. You must solve this problem by rearranging the links of the list.

```
   public class LinkedIntList {
      private ListNode front;
      ...
   }

   public class ListNode {
      public int data;
      public ListNode next;
      ...
   }

public void switchPairs() {
   if (front != null && front.next != null) {
      ListNode current = front.next;
      front.next = current.next;
      current.next = front;
      front = current;
      current = current.next;

      while (current.next != null && current.next.next != null) {
         ListNode temp = current.next.next;
         current.next.next = temp.next;
```

```
            temp.next = current.next;
            current.next = temp;
            current = temp.next;
        }
    }
}
```

Exercise 16.9: stutter

Write a method stutter that doubles the size of a list by replacing every integer in the list with two of that integer. For example, suppose a variable list stores the following sequence of integers:

[1, 8, 19, 4, 17]

After a call of list.stutter(), it should store the following sequence of integers:

[1, 1, 8, 8, 19, 19, 4, 4, 17, 17]

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
    private ListNode front;   // null for an empty list
    ...
}

public void stutter() {
    ListNode current = front;
    while (current != null) {
        current.next = new ListNode(current.data, current.next);
        current = current.next.next;
    }
}
```

Exercise 16.10: stretch

Write a method stretch that takes an integer *n* as a parameter and that increases a list of integers by a factor of *n* by replacing each integer in the original list with *n* copies of that integer. For example, if a variable called list stores this sequence of values:

[18, 7, 4, 24, 11]

And we make the following call:

list.stretch(3);

The list should store the following values after the method is called:

[18, 18, 18, 7, 7, 7, 4, 4, 4, 24, 24, 24, 11, 11, 11]

If the value of *n* is less than or equal to 0, the list should be empty after the call.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...

   // your code goes here
}

public void  stretch(int copies) {
   if (copies <= 0) {
      front = null;
   } else {
      ListNode current = front;

      while (current != null) {
         int value = current.data;

         for (int i = 1; i < copies; i++) {
            current.next = new ListNode(value, current.next);
            current = current.next;
         }

         current = current.next;
      }
   }
}
```

Exercise 16.11: compress

Write a method compress that could be added to the LinkedIntList class, that accepts an integer *n* representing a "compression factor" and replaces every *n* elements with a single element whose data value is the sum of those *n* nodes. Suppose a LinkedIntList variable named list stores the following values:

[2, 4, 18, 1, 30, -4, 5, 58, 21, 13, 19, 27]

If you made the call of list.compress(2);, the list would replace every two elements with a single element (2 + 4 = 6, 18 + 1 = 19, 30 + (-4) = 26, ...), storing the following elements:

[6, 19, 26, 63, 34, 46]

If you then followed this with a second call of list.compress(3);, the list would replace every three elements with a single element (6 + 19 + 26 = 51, 63 + 34 + 46 = 143), storing the following elements:

[51, 143]

If the list's size is not an even multiple of *n*, whatever elements are left over at the end are compressed into one node. For example, the original list on this page contains 12 elements, so if you made a call on it of list.compress(5);, the list would compress every five elements, (2 + 4 + 18 + 1 + 30 = 55, -4 + 5 + 58 + 21 + 13 = 93), with the last two leftover elements compressing into a final third element (19 + 27 = 46), resulting in the following list:

[55, 93, 46]

If *n* is greater than or equal to the list size, the entire list compresses into a single element. If the list is empty, the result after the call is empty regardless of what factor *n* is passed. You may assume that the value passed for *n* is >= 1.

For full credit, you may not create any new ListNode objects, though you may have as many ListNode variables as you like. For full credit, your solution must also run in O(*n*) time. Assume that you are adding this method to the LinkedIntList class below. You may not call any other methods of the class.

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void compress(int factor) {
   ListNode current = front;
   while (current != null) {
      int i = 1;
      ListNode current2 = current.next;
      while (current2 != null && i < factor) {
         current.data += current2.data;
         current.next = current.next.next;
         i++;
         current2 = current2.next;
      }
      current = current.next;
```

```
    }
}
```

Exercise 16.12: split

Write a method split that rearranges the elements of a list so that all of the negative values appear before all of the non-negatives. For example, suppose a variable list stores the following sequence of values:

[8, 7, -4, 19, 0, 43, -8, -7, 2]

The call of list.split(); should rearrange the list to put the negatives first. One possible arrangement would be the following:

[-4, -8, -7, 8, 7, 19, 0, 43, 2]

But it matters only that the negatives appear before the non-negatives. So this is only one possible solution. Another legal solution would be to rearrange the values this way:

[-7, -8, -4, 2, 43, 0, 19, 7, 8]

You are not allowed to swap data fields or to create any new nodes to solve this problem; you must rearrange the list by rearranging the links of the list. You also may not use auxiliary structures like arrays, ArrayLists, stacks, queues, etc, to solve this problem.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
    private ListNode front;   // null for an empty list
    ...
}

public void split() {
    if (front != null) {
        ListNode current = front;
        while (current.next != null) {
            if (current.next.data < 0) {
                ListNode temp = current.next;
                current.next = current.next.next;
                temp.next = front;
                front = temp;
            } else {
                current = current.next;
            }
        }
    }
}
```

Exercise 16.13: transerFrom

Write a method transferFrom that accepts a second LinkedIntList as a parameter and that moves values from the second list to this list. You are to attach the second list's elements to the end of this list. You are also to empty the second list. For example, suppose two lists store these sequences of values:

```
list1: [8, 17, 2, 4]
list2: [1, 2, 3]
```

The call of list1.transferFrom(list2); should leave the lists as follows:

```
list1: [8, 17, 2, 4, 1, 2, 3]
list2: []
```

The order of the arguments matters; list2.transferFrom(list1); would have left the lists as follows:

```
list1: []
list2: [1, 2, 3, 8, 17, 2, 4]
```

Either of the two lists could be empty, but you can assume that neither list is null. You are not to create any new nodes. Your method should simply change links of the lists to join them together.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
    private ListNode front;   // null for an empty list
    ...
}

public void transferFrom(LinkedIntList other) {
    if (front == null) {
        front = other.front;
    } else {
        ListNode current = front;
        while (current.next != null) {
            current = current.next;
        }
        current.next = other.front;
    }
    other.front = null;
}
```

Exercise 16.14: removeAll
Write a method removeAll that removes all occurrences of a particular value. For example, if a variable list contains the following values:

[3, 9, 4, 2, 3, 8, 17, 4, 3, 18]

The call of list.removeAll(3); would remove all occurrences of the value 3 from the list, yielding the following values:

[9, 4, 2, 8, 17, 4, 18]

If the list is empty or the value doesn't appear in the list at all, then the list should not be changed by your method. You must preserve the original order of the elements of the list.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void removeAll(int value) {
   while (front != null && front.data == value) {
      front = front.next;
   }
   if (front != null) {
      ListNode current = front;
      while (current.next != null) {
         if (current.next.data == value) {
            current.next = current.next.next;
         } else {
            current = current.next;
         }
      }
   }
}
```

Exercise 16.15: equals
Write a method equals2 that accepts a second list as a parameter and that returns true if the two lists are equal and that returns false otherwise. Two lists are considered equal if they store exactly the same values in exactly the same order and have exactly the same length. (Note: On

the original section handout, this method is called equals; but Practice-It already defines an equals method for LinkedIntList for internal use, so we must call your method equals2 here to avoid a conflict.)

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}
```

```
public boolean equals2(LinkedIntList other) {
   ListNode current1 = front;
   ListNode current2 = other.front;
   while (current1 != null && current2 != null) {
      if (current1.data != current2.data) {
         return false;
      }
      current1 = current1.next;
      current2 = current2.next;
   }
   return current1 == null && current2 == null;
}
```

Exercise 16.16: removeEvens

Write a method removeEvens that removes the values in even-numbered indexes from a list, returning a new list containing those values in their original order. For example, if a variable list1 stores these values:

list1: [8, 13, 17, 4, 9, 12, 98, 41, 7, 23, 0, 92]

And the following call is made:

```
LinkedIntList list2 = list1.removeEvens();
```

After the call, list1 and list2 should store the following values:

list1: [13, 4, 12, 41, 23, 92]
list2: [8, 17, 9, 98, 7, 0]

Notice that the values stored in list2 are the values that were originally in even-valued positions (index 0, 2, 4, etc.) and that these values appear in the same order as in the original list. Also notice that the values left in list1 also appear in their original relative order. Recall that

LinkedIntList has a zero-argument constructor that returns an empty list. You may not call any methods of the class other than the constructor to solve this problem. You are not allowed to create any new nodes or to change the values stored in data fields to solve this problem; You must solve it by rearranging the links of the list.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public LinkedIntList removeEvens() {
   LinkedIntList result = new LinkedIntList();
   if (front != null) {
      result.front = front;
      front = front.next;
      ListNode current = front;
      ListNode resultLast = result.front;
      while (current != null && current.next != null) {
         resultLast.next = current.next;
         resultLast = current.next;
         current.next = current.next.next;
         current = current.next;
      }
      resultLast.next = null;
   }
   return result;
}
```

Exercise 16.17: removeRange

Write a method removeRange that accepts a starting and ending index as parameters and removes the elements at those indexes (inclusive) from the list. For example, if a variable list stores the following values:

[8, 13, 17, 4, 9, 12, 98, 41, 7, 23, 0, 92]

And the following call is made:

listRange.removeRange(3, 8);

Then the values between index 3 and index 8 (the value 4 and the value 7) are removed, leaving the following list:

[8, 13, 17, 23, 0, 92]

You should throw an IllegalArgumentException if either of the positions is negative. Otherwise you may assume that the positions represent a legal range of the list (0 <= start index <= end index < size of list).

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}
```

```
public void removeRange(int low, int high) {
   if (low < 0 || high < 0) {
      throw new IllegalArgumentException();
   }
   if (low == 0) {
      while (high >= 0) {
         front = front.next;
         high--;
      }
   } else {
      ListNode current = front;
      int count = 1;
      while (count < low) {
         current = current.next;
         count++;
      }
      ListNode current2 = current.next;
      while (count < high) {
         current2 = current2.next;
         count++;
      }
      current.next = current2.next;
   }
}
```

Exercise 16.18: doubleList
Write a method doubleList that doubles the size of a list by appending a copy of the original sequence to the end of the list. For example, if a variable list stores this sequence of values:

[1, 3, 2, 7]

And we make the call of list.doubleList(); then it should store the following values after the call:

[1, 3, 2, 7, 1, 3, 2, 7]

Notice that it has been doubled in size by having the original sequence appear two times in a row. You may not make assumptions about how many elements are in the list. You may not call any methods of the class to solve this problem. If the original list contains n nodes, then you should construct exactly n nodes to be added. You may not use any auxiliary data structures to solve this problem (no array, ArrayList, stack, queue, String, etc). Your method should run in O(n) time where n is the number of nodes in the list.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void doubleList() {
   if (front != null) {
      ListNode half2 = new ListNode(front.data);
      ListNode back = half2;
      ListNode current = front;
      while (current.next != null) {
         current = current.next;
         back.next = new ListNode(current.data);
         back = back.next;
      }
      current.next = half2;
   }
}
```

Exercise 16.19: rotate

Write a method rotate that moves the value at the front of a list of integers to the end of the list. For example, if a variable called list stores the following sequence of values:

[8, 23, 19, 7, 45, 98, 102, 4]

Then the call of list.rotate(); should move the value 8 from the front of the list to the back of the list, yielding this sequence of values:

[23, 19, 7, 45, 98, 102, 4, 8]

The other values in the list should retain the same order as in the original list. If the method is called for a list of 0 or 1 elements it should have no effect on the list. You are not allowed to construct any new nodes to solve this problem and you are not allowed to change any of the integer values stored in the nodes. You must solve the problem by rearranging the links of the list.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void rotate() {
   if (front != null && front.next != null) {
      ListNode temp = front;
      front = front.next;
      ListNode current = front;
      while (current.next != null) {
         current = current.next;
      }
      current.next = temp;
      temp.next = null;
   }
}
```

Exercise 16.20: shift

Write a method shift that rearranges the elements of a list of integers by moving to the end of the list all values that are in odd-numbered positions and otherwise preserving list order. For example, suppose a variable list stores the following values:

[0, 1, 2, 3, 4, 5, 6, 7]

The call of list.shift(); should rearrange the list to be:

[0, 2, 4, 6, 1, 3, 5, 7]

In this example the values in the original list were equal to their positions and there were an even number of elements, but that won't necessarily be the case. For example, if list had instead stored the following:

[4, 17, 29, 3, 8, 2, 28, 5, 7]

Then after the call list.shift(); the list would store:

[4, 29, 8, 28, 7, 17, 3, 2, 5]

Notice that it doesn't matter whether the value itself is odd or even. What matters is whether the value appears in an odd index (index 1, 3, 5, etc). Also notice that the original order of the list is otherwise preserved. You may not construct any new nodes and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes; you must solve this problem by rearranging the links of the list.

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void shift() {
   if (front != null && front.next != null) {
      ListNode otherFront = front.next;
      front.next = front.next.next;
      ListNode current1 = front;
      ListNode current2 = otherFront;
      while (current1.next != null) {
         current1 = current1.next;
         if (current1.next != null) {
            current2.next = current1.next;
            current1.next = current1.next.next;
            current2 = current2.next;
         }
      }
      current2.next = null;
      current1.next = otherFront;
   }
}
```

Exercise 16.21: reverse

Write a method reverse that reverses the order of the elements in the list. For example, if the variable list initially stores this sequence of integers:

[1, 8, 19, 4, 17]

It should store the following sequence of integers after reverse is called:

[17, 4, 19, 8, 1]

Assume that you are adding this method to the LinkedIntList class as defined below:

```
public class LinkedIntList {
   private ListNode front;   // null for an empty list
   ...
}

public void reverse() {
   ListNode current = front;
   ListNode previous = null;
   while (current != null) {
      ListNode nextNode = current.next;
      current.next = previous;
      previous = current;
      current = nextNode;
   }
   front = previous;
}
```

Chapter 17: Binary Tree

Exercise 17.1: countLeftNodes

Write a method countLeftNodes that returns the number of left children in the tree. A left child is a node that appears as the root of the left-hand subtree of another node. An empty tree has 0 left nodes. For example, the following tree has four left children (the nodes storing the values 5, 1, 4, and 7):

```
         +---+
         | 3 |
         +---+
        /     \
    +---+     +---+
    | 5 |     | 2 |
    +---+     +---+
   /         /     \
+---+     +---+     +---+
| 1 |     | 4 |     | 6 |
+---+     +---+     +---+
        /
    +---+
    | 7 |
    +---+
```

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public int countLeftNodes() {
   return countLeftNodes(overallRoot);
}
private int countLeftNodes(IntTreeNode root) {
   if (root == null) {
      return 0;
   } else if (root.left == null) {
      return countLeftNodes(root.right);
   } else {
      return 1 + countLeftNodes(root.left) + countLeftNodes(root.right);
   }
}
```

Exercise 17.2: countEmpty

Write a method countEmpty that returns the number of empty branches in a tree. An empty tree is considered to have one empty branch (the tree itself). For non-empty trees, your method(s) should count the total number of empty branches among the nodes of the tree. A leaf node has two empty branches. A node with one non-empty child has one empty branch. A node with two non-empty children (a full branch) has no empty branches. For example the tree below has 15 empty branches (indicated by circles):

```
                +---+
                | 0 |
                +---+
              /    \
            +---+     +---+
            | 4 |     | 6 |
            +---o     +---+
          /         /     \
        +---+     +---+     +---+
        | 3 |     | 0 |     | 1 |
        +---o     +---+     o---+
       /        /     \          \
    +---+     +---+     +---+     +---+
    | 4 |     | 3 |     | 8 |     | 5 |
    +---o     o---+     o---o     +---+
   /               \      /    \
+---+             +---+     +---+     +---+
| 1 |             | 9 |     | 2 |     | 7 |
o---o             o---o     o---o     o---o
```

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public int countEmpty() {
   return countEmpty(overallRoot);
}
private int countEmpty(IntTreeNode root) {
   if (root == null) {
      return 1;
   } else {
      return countEmpty(root.left) + countEmpty(root.right);
   }
}
```

Exercise 17.3: depthSum

Write a method depthSum that returns the sum of the values stored in a binary tree of integers weighted by the depth of each value. You should return the value at the overallRoot plus 2 times the values stored at the next level of the tree plus 3 times the values stored at the next level of the tree plus 4 times the values stored at the next level of the tree and so on. For example, in the tree below:

```
      +---+
      | 9 |
      +---+
     /   \
   +---+   +---+
   | 7 |   | 6 |
   +---+   +---+
  /   \     \
+---+   +---+   +---+
| 3 |   | 2 |   | 4 |
+---+   +---+   +---+
     /         \
   +---+         +---+
   | 5 |         | 2 |
   +---+         +---+
```

The sum would be computed as:

1 * 9 + 2 * (7 + 6) + 3 * (3 + 2 + 4) + 4 * (5 + 2) = 90

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public int depthSum() {
   return depthSum(overallRoot, 1);
}
private int depthSum(IntTreeNode root, int depth) {
   if (root == null) {
      return 0;
   } else {
      return depth * root.data + depthSum(root.left, depth + 1)
                               + depthSum(root.right, depth + 1);
```

```
    }
}
```

Exercise 17.4: countEvenBranches

Write a method countEvenBranches that returns the number of branch nodes in a binary tree that contain even numbers. A branch node is one that has one or two children (i.e., not a leaf node). An empty tree has 0 even branches. For example, if a variable tree stores a reference to the following tree:

```
         +---+
         | 2 |
         +---+
        /     \
    +---+     +---+
    | 8 |     | 1 |
    +---+     +---+
   /         /     \
+---+     +---+     +---+
| 0 |     | 7 |     | 6 |
+---+     +---+     +---+
         /             \
    +---+              +---+
    | 4 |              | 9 |
    +---+              +---+
```

Then the call tree.countEvenBranches(); should return 3 because there are three branch nodes with even values (2, 8, and 6). Notice that the leaf nodes with even values are not included (the nodes storing 0 and 4).

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public int countEvenBranches() {
   return countEvenBranches(overallRoot);
}
private int countEvenBranches(IntTreeNode root) {
   if (root == null) {
      return 0;
   } else if (root.left == null && root.right == null) {
      return 0;
```

```
    } else if (root.data % 2 == 0) {
        return 1 + countEvenBranches(root.left) + countEvenBranches(root.right);
    } else {
        return countEvenBranches(root.left) + countEvenBranches(root.right);
    }
}
```

Exercise 17.5: printLevel

Write a method printLevel that accepts an integer parameter *n* and that prints the values at level *n* from the left to right, one per line. We will use the convention that the overallRoot is at level 1, that its children are at level 2, and so on. For example, if a variable tree stores a reference to the following tree:

```
            +----+
            | 12 |
            +----+
           /      \
          /        \
      +----+      +----+
      | 19 |      | 93 |
      +----+      +----+
     /      \          \
    /        \          \
+----+      +----+      +----+
| 11 |      | 14 |      | 15 |
+----+      +----+      +----+
           /
          /
      +----+
      | 10 |
      +----+
```

Then the call tree.printLevel(3); would produce the following output:

```
11
14
15
```

If there are no values at the level, your method should produce no output. Your method should throw an IllegalArgumentException if passed a value for a level *n* that is less than 1.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public void printLevel(int target) {
   if(target < 1) {
      throw new IllegalArgumentException();
   }
   printLevel(overallRoot, target, 1);
}
private void printLevel(IntTreeNode root, int target, int level) {
   if(root != null) {
      if(level == target) {
         System.out.println(root.data);
      } else {
         printLevel(root.left, target, level + 1);
         printLevel(root.right, target, level + 1);
      }
   }
}
```

Exercise 17.6: printLeaves

Write a method printLeaves that outputs the leaves of a binary tree from right to left. More specifically, the leaves should be printed in the reverse order that they would be printed using any of the standard traversals. For example, if a variable tree stores a reference to the following tree:

```
      +---+
      | 2 |
      +---+
     /   \
   +---+   +---+
   | 8 |   | 1 |
   +---+   +---+
   /     /  \
+---+   +---+   +---+
| 0 |   | 7 |   | 6 |
+---+   +---+   +---+
     /         \
   +---+          +---+
   | 4 |          | 9 |
   +---+          +---+
```

Then the call of t.printLeaves(); should produce the following output:

```
leaves: 9 4 0
```

If the tree does not have any leaves (an empty tree), simply print:

```
no leaves
```

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public void printLeaves() {
   if (overallRoot == null) {
      System.out.println("no leaves");
   } else {
      System.out.print("leaves:");
      printLeaves(overallRoot);
      System.out.println();
   }
}
private void printLeaves(IntTreeNode root) {
   if (root != null) {
      if (root.left == null && root.right == null) {
         System.out.print(" " + root.data);
      } else {
         printLeaves(root.right);
         printLeaves(root.left);
      }
   }
}
```

Exercise 17.7: isFull

Write a method isFull that returns whether or not a binary tree is full (true, if it is, false if otherwise). A full binary tree is one in which every node has 0 or 2 children. Below are examples of each.

full tree	**not a full tree**
+---+ | 2 |	+---+ | 7 |

```
        +---+
       /     \
    +---+    +---+
    | 3 |    | 1 |
    +---+    +---+
   /    \
+---+    +---+
| 8 |    | 7 |
+---+    +---+
```

```
        +---+
       /     \
    +---+    +---+
    | 4 |    | 0 |
    +---+    +---+
   /    \        \
+---+    +---+    +---+
| 9 |    | 2 |    | 8 |
+---+    +---+    +---+
```

By definition, the empty tree is considered full.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public boolean isFull() {
   return (overallRoot == null || isFull(overallRoot));
}
private boolean isFull(IntTreeNode root) {
   if(root.left == null && root.right == null) {
      return true;
   } else {
      return (root.left != null && root.right != null &&
         isFull(root.left) && isFull(root.right));
   }
}
```

Exercise 17.8: toString

Write a method toString2 for a binary tree of integers. (On your section handout the method is called toString, but Practice-It needs you to call your method toString2 because toString is already used for another purpose.) The method should return "empty" for an empty tree. For a leaf node, it should return the data in the node as a String. For a branch node, it should return a parenthesized String that has three elements separated by commas:

1. The data at the root.
2. A String representation of the left subtree.
3. A String representation of the right subtree.

For example, if a variable tree stores a reference to the following tree:

```
      +---+
      | 2 |
      +---+
     /    \
   +---+   +---+
   | 8 |   | 1 |
   +---+   +---+
  /       /   \
+---+   +---+   +---+
| 0 |   | 7 |   | 6 |
+---+   +---+   +---+
     /         \
   +---+         +---+
   | 4 |         | 9 |
   +---+         +---+
```

Then the call tree.toString2(); should return the following String:

"(2, (8, 0, empty), (1, (7, 4, empty), (6, empty, 9)))"

The quotes above are used to indicate that this is a String but should not be included in the String you return.

(Note: On the original section handout, this method is called toString, but Practice-It already defines a toString method for IntTree for use in other problems, so this method must be called toString2 to avoid a conflict.)

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public String toString2() {
   return toString(overallRoot);
}
private String toString(IntTreeNode root) {
   if (root == null) {
      return "empty";
   } else if (root.left == null && root.right == null) {
      return "" + root.data;
```

```
    } else {
        return "(" + root.data + ", " + toString(root.left) +
                ", " + toString(root.right) + ")";
    }
}
```

Exercise 17.9: equals

Write a method equals that could be added to the IntTree class. (On your handout this method is called "equals", but Practice-It needs to use the name "equals" for another purpose, so we'll call it "equals2" here.) The method accepts another binary tree of integers as a parameter and compares the two trees to see if they are equal to each other. For example, if variables of type IntTree called t1 and t2 have been initialized, then the call of t1.equals2(t2) will return true if the trees are equal and false if otherwise.

Two trees are considered equal if they have exactly the same structure and store the same values. Each node in one tree must have a corresponding node in the other tree in the same location relative to the root and storing the same value. Two empty trees are considered equal to each other.

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc. Your method should not change the structure or contents of either of the two trees being compared.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
}

public boolean equals2(IntTree other) {
    return equals(this.overallRoot, other.overallRoot);
}
private boolean equals2(IntTreeNode root1, IntTreeNode root2) {
    if (root1 == null || root2 == null) {
        return root1 == null && root2 == null;
    } else {
        return root1.data == root2.data
                && equals(root1.left, root2.left)
                && equals(root1.right, root2.right);
    }
}
```

Exercise 17.10: doublePositives

Write a method doublePositives that doubles all data values greater than 0 in a binary tree of integers. For example, the before and after of a call to doublePositives on a sample tree are shown below:

```
          Before Call                        After Call
-----------------------------------------------------------------
          +----+                             +----+
          | -9 |                             | -9 |
          +----+                             +----+
         /      \                           /      \
        /        \                         /        \
    +----+      +----+                 +----+      +----+
    |  3 |      | 15 |                 |  6 |      | 30 |
    +----+      +----+                 +----+      +----+
   /           /      \               /           /      \
  /           /        \             /           /        \
+----+    +----+    +----+         +----+    +----+    +----+
|  0 |    | 12 |    | 24 |         |  0 |    | 24 |    | 48 |
+----+    +----+    +----+         +----+    +----+    +----+
         /      \                           /      \
        /        \                         /        \
    +----+      +----+                 +----+      +----+
    |  6 |      | -3 |                 | 12 |      | -3 |
    +----+      +----+                 +----+      +----+
```

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
}

public void doublePositives() {
    doublePositives(overallRoot);
}
private void doublePositives(IntTreeNode root) {
    if (root != null) {
        if (root.data > 0) {
            root.data *= 2;
        }
        doublePositives(root.left);
        doublePositives(root.right);
    }
}
```

Exercise 17.11: numberNodes
Write a method numberNodes that changes the data stored in a binary tree, assigning sequential integers starting with 1 to each node so that a pre-order traversal will produce the numbers in order(1, 2, 3, etc.). For example, given the tree referenced by tree below at left, the call of tree.numberNodes(); would overwrite the existing data assigning the nodes values from 1 to 6 so that a pre-order traversal of the tree would produce 1, 2, 3, 4, 5, 6.

Before Call	**After Call**
`          +---+`	`          +---+`
`	7	`	`	1	`
`          +---+`	`          +---+`
`         /     \`	`         /     \`
`     +---+   +---+`	`     +---+   +---+`
`	3		9	`	`	2		5	`
`     +---+   +---+`	`     +---+   +---+`
`    /    \        \`	`    /    \        \`
`+---+   +---+   +---+`	`+---+   +---+   +---+`
`| 9		2		0 |`	`| 3		4		6 |`
`+---+   +---+   +---+`	`+---+   +---+   +---+`

You are not to change the structure of the tree. You are simply changing the values stored in the data fields. Your method should return a count of how many nodes were in the tree.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
}

public int numberNodes() {
    return numberNodes(overallRoot, 1);
}
private int numberNodes(IntTree.IntTreeNode root, int count) {
    if (root == null) {
        return 0;
    } else {
        root.data = count;
        int leftNum  = numberNodes(root.left,  count + 1);
        int rightNum = numberNodes(root.right, count + 1 + leftNum);
        return leftNum + rightNum + 1;
    }
}
```

Exercise 17.12: removeLeaves

Write a method removeLeaves that removes the leaves from a tree. A leaf node that has empty left and right subtrees. If a variable tree refers to the first tree below, the call of tree.removeLeaves(); should remove the four leaves from the tree (the nodes with data values 1, 4, 6, and 0) leaving the next tree shown below.

A second call on the method would eliminate the two leaves in this tree (the nodes with data values 3 and 8), shown in the third tree below. A third call would eliminate the one leaf with data value 9, and a fourth call would leave an empty tree because the previous tree was exactly one leaf node. If your method is called on an empty tree, the method does not change the tree because there are no nodes of any kind (leaf or not).

Call	#	Tree

(before call)	0	`       +---+`
		`       | 7 |`
		`   ___ +---+ ___`
		`  /           \`
		`+---+       +---+`
		`| 3 |       | 9 |`
		`+---+       +---+`
		`/   \       /   \`
		`+---+ +---+ +---+ +---+`
		`| 1 | | 4 | | 6 | | 8 |`
		`+---+ +---+ +---+ +---+`
		`                  \`
		`                  +---+`
		`                  | 0 |`
		`                  +---+`

tree.removeLeaves();	1	`       +---+`
		`       | 7 |`
		`       +---+`
		`       /   \`

		`+---+    +---+`
		`| 3 |    | 9 |`
		`+---+    +---+`
		`            \`
		`           +---+`
		`           | 8 |`
		`           +---+`

	2	`+---+`
		`| 7 |`
		`+---+`
		`    \`
		`    +---+`
		`    | 9 |`
		`    +---+`

	3	`+---+`
		`| 7 |`
		`+---+`

	4	null

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}


public void removeLeaves() {
   overallRoot = removeLeaves(overallRoot);
}
private IntTreeNode removeLeaves(IntTreeNode root) {
   if(root != null) {
      if(root.left == null && root.right == null) {
         root = null;
      } else {
         root.left  = removeLeaves(root.left);
         root.right = removeLeaves(root.right);
      }
   }
   return root;
}
```

Exercise 17.14: completeToLevel

Write a method completeToLevel that accepts an integer *n* as a parameter and that adds nodes to a tree so that the first *n* levels are complete. A level is complete if every possible node at that level is not null. We will use the convention that the overall root is at level 1, it's children are at level 2, and so on. You should preserve any existing nodes in the tree. Any new nodes added to the tree should have -1 as their data.

For example, if a variable called tree refers to the tree below at left and you make the call of tree.completeToLevel(3);, the variable tree should store the tree below at right after the call.

```
      Before Call        |            After Call
-------------------------+-----------------------------------
        +----+           |              +----+
        | 17 |           |              | 17 |
        +----+           |            _ +----+ _
        /    \           |          _ /        \ _
       /      \          |           /          \
    +----+   +----+      |     +----+           +----+
    | 83 |   |  6 |      |     | 83 |           |  6 |
    +----+   +----+      |     +----+           +----+
    /            \       |      /   \           /   \
   /              \      |     /     \         /     \
+----+          +----+   | +----+   +----+   +----+   +----+
| 19 |          | 87 |   | | 19 |   | -1 |   | -1 |   | 87 |
+----+          +----+   | +----+   +----+   +----+   +----+
     \          /        |    \                   /
      \        /         |     \                 /
    +----+   +----+      |   +----+           +----+
    | 48 |   | 75 |      |   | 48 |           | 75 |
    +----+   +----+      |   +----+           +----+
```

In this case, the request was to fill in nodes as necessary to ensure that the first 3 levels are complete. There were two nodes missing at level 3, Notice that level 4 of this tree is not complete because the call requested that nodes be filled in to level 3 only.

Keep in mind that your method might need to fill in several different levels. Your method should throw an IllegalArgumentException if passed a value for level that is less than 1.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
    private IntTreeNode overallRoot;
    ...
}

public void completeToLevel(int target) {
    if (target < 1) {
        throw new IllegalArgumentException();
    }
    overallRoot = complete(overallRoot, target, 1);
}
private IntTreeNode complete(IntTreeNode root, int target, int level) {
    if (level <= target) {
        if (root == null) {
            root = new IntTreeNode(-1);
        }
        root.left  = complete(root.left,  target, level + 1);
        root.right = complete(root.right, target, level + 1);
    }
    return root;
}
```

Exercise 17.15: trim

Exercise 17.16: tighten

Write a method trim that could be added to the IntTree class. The method accepts minimum and maximum integers as parameters and removes from the tree any elements that are not within that range, inclusive. For this method you should assume that your tree is a binary search tree (BST) and that its elements are in valid BST order. Your method should maintain the BST ordering property of the tree.

For example, suppose a variable of type IntTree called tree stores the following elements:

tree

```
                        +----+
                        | 50 |
                   _____ +----+ _____
                  /              \
              +----+            +----+
              | 38 |            | 90 |
          _____ +----+          ___ +----+
         /       \           /
      +----+    +----+   +----+
     | 14 |        | 42 |   | 54 |
     +----+        +----+   +----+
    /      \                  \
 +----+  +----+                +----+
 |  8 |  | 20 |                | 72 |
 +----+  +----+                +----+
            \                 /    \
          +----+          +----+  +----+
          | 26 |          | 61 |  | 83 |
          +----+          +----+  +----+
```

The table below shows what the state of the tree would be if various trim calls were made. The calls shown are separate; it's not a chain of calls in a row. You may assume that the minimum is less than or equal to the maximum.

```
  tree.trim(25, 72);        tree.trim(54, 80);   tree.trim(18, 42);   tree.trim(3, 7);

        +----+               +----+                   +----+
        | 50 |               | 54 |                   | 38 |
       _+----+               +----+                  _+----+
      /       \                  \                  /       \
  +----+       +----+            +----+        +----+      +----+
  | 38 |       | 54 |            | 72 |        | 20 |      | 42 |
  +----+       +----+            +----+        +----+      +----+
  /    \            \            /                 \
+----+ +----+      +----+    +----+               +----+
| 26 | | 42 |      | 72 |    | 61 |               | 26 |
+----+ +----+      +----+    +----+               +----+
                   /
              +----+
              | 61 |
              +----+
```

Hint: The BST ordering property is important for solving this problem. If a node's data value is too large or too small to fit within the range, this may also tell you something about whether that node's left or right subtree elements can be within the range. Taking advantage of such information makes it more feasible to remove the correct nodes.

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public void trim(int min, int max) {
   overallRoot = trim(overallRoot, min, max);
}
private IntTreeNode trim(IntTreeNode root, int min, int max) {
   if (root != null) {
      if (root.data < min) {
         root = trim(root.right, min, max);
      } else if (root.data > max) {
         root = trim(root.left, min, max);
      } else {
         root.left  = trim(root.left,  min, max);
         root.right = trim(root.right, min, max);
      }
   }
   return root;
}
```

Exercise 17.17: combineWith

Write a method tighten that eliminates branch nodes that have only one child. For example, if a variable called tree stores the tree below at left, the call of tree.tighten(); should leave tree storing the tree at right:

```
            Before Call                          After Call
 ------------------------------------+------------------------------------
              +---+                  |            +---+
              | 2 |                  |            | 2 |
          ___ +---+ ___              |        ___ +---+ ___
         /           \               |       /           \
      +---+           +---+          |    +---+           +---+
      | 8 |           | 9 |          |    | 7 |           | 0 |
      +---+           +---+          |    +---+           +---+
      /               /              |    /   \           /   \
   +---+           +---+             | +---+  +---+  +---+   +---+
   | 7 |           | 6 |             | | 4 |  | 3 |  | 4 |   | 5 |
   +---+           +---+             | +---+  +---+  +---+   +---+
   /   \               \             |
+---+   +---+           +---+
| 4 |   | 1 |           | 0 |
+---+   +---+           +---+
          \            /     \
          +---+    +---+     +---+
          | 3 |    | 4 |     | 5 |
          +---+    +---+     +---+
```

The nodes that stored the values 8, 9, 6, and 1 have been eliminated because each had one child. When a node is removed, it is replaced by its child. This can lead to multiple replacements because the child might itself be replaced (as in the case of 9 which is replaced by 6 which is replaced by 0).

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public void tighten() {
   overallRoot = tighten(overallRoot);
}
private IntTreeNode tighten(IntTreeNode root) {
   if (root != null) {
      root.left = tighten(root.left);
      root.right = tighten(root.right);
      if (root.left != null && root.right == null) {
         root = root.left;
      } else if (root.left == null && root.right != null) {
         root = root.right;
      }
   }
   return root;
}
```

Exercise 17.18: inOrderList

Write a method combineWith that could be added to the IntTree class. The method accepts another binary tree of integers as a parameter and combines the two trees into a new third tree which is returned. The new tree's structure should be a union of the structures of the two original trees. It should have a node in any location where there was a node in either of the original trees (or both). The nodes of the new tree should store an integer indicating which of the original trees had a node at that position (1 if just the first tree had the node, 2 if just the second tree had the node, 3 if both trees had the node).

For example, suppose IntTree variables t1 and t2 have been initialized and store the following trees:

```
                t1                              t2
            +----+                          +----+
            |  9 |                          |  0 |
            +----+                          +----+
           /      \                        /      \
          /        \                      /        \
     +----+      +----+              +----+      +----+
     |  6 |      | 14 |              | -3 |      |  8 |
     +----+      +----+              +----+      +----+
    /     \           \             /           /     \
   /       \           \           /           /       \
+----+    +----+    +----+    +----+    +----+    +----+
|  9 |    |  2 |    | 11 |    |  8 |    |  5 |    |  6 |
+----+    +----+    +----+    +----+    +----+    +----+
          /                                 \
         /                                   \
     +----+                                +----+
     |  4 |                                |  1 |
     +----+                                +----+
```

Then the following call:

IntTree t3 = t1.combineWith(t2);

Will return a reference to the following tree:

```
            t3
        +---+
        | 3 |
    ___ +---+ ___
   /             \
+---+           +---+
| 3 |           | 3 |
    +---+           +---+
   /     \         /     \
+---+   +---+   +---+   +---+
| 3 |   | 1 |   | 2 |   | 3 |
+---+   +---+   +---+   +---+
      /             \
    +---+           +---+
    | 1 |           | 2 |
    +---+           +---+
```

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, lists, etc. Your method should not change the structure or contents of either of the two trees being compared.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public IntTree combineWith(IntTree other) {
   IntTree result = new IntTree();
   result.overallRoot = combine(this.overallRoot, other.overallRoot);
   return result;
}
private IntTreeNode combine(IntTreeNode root1, IntTreeNode root2) {
   if (root1 == null) {
      if (root2 == null) {
         return null;
      } else {
         return new IntTreeNode(2, combine(null, root2.left),
                                combine(null, root2.right));
      }
   } else {
      if (root2 == null) {
         return new IntTreeNode(1, combine(root1.left, null),
                                combine(root1.right, null));
      } else {
         return new IntTreeNode(3, combine(root1.left, root2.left),
                                combine(root1.right, root2.right));
      }
   }
}
```

Exercise 17.19: evenLevels

Write a method inOrderList that could be added to the IntTree class. The method returns a list containing the sequence of values obtained from an in-order traversal of your binary tree of integers. For example, if a variable tree refers to the following tree:

```
          +---+
          | 7 |
          +---+
         /     \
     +---+     +---+
     | 4 |     | 2 |
     +---+     +---+
    /         /     \
+---+     +---+     +---+
| 9 |     | 5 |     | 0 |
+---+     +---+     +---+
```

Then the call tree.inOrderList() should return the following list:

[9, 4, 7, 5, 2, 0]

If the tree is empty, your method should return an empty list.

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class nor create any data structures such as arrays, etc. other than the list you will eventually return. Your method should not change the structure or contents of the tree.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public List<Integer> inOrderList() {
   List<Integer> result = new ArrayList<Integer>();
   inOrderList(overallRoot, result);
   return result;
}
private void inOrderList(IntTreeNode root, List<Integer> result) {
   if (root != null) {
        inOrderList(root.left, result);
        result.add(root.data);
        inOrderList(root.right, result);
    }
}
```

Exercise 17.20: makePerfect
Write a method evenLevels that could be added to the IntTree class from lecture and section. The method should make sure that all branches end on an even level. If a leaf node is on an odd level it should be removed from the tree. We will define the root as being on level 1.

The following table shows the results of a call of your method on a particular tree:

```
                before                 |              after
---------------------------------------+------------------------------
              +----+                   |          +----+
              | 67 |                   |          | 67 |
              +----+                   |          +----+
            /       \                  |        /       \
          /           \                |      /           \
      +----+           +----+          |  +----+           +----+
      | 80 |           | 52 |          |  | 80 |           | 52 |
      +----+           +----+          |  +----+           +----+
      /    \         /    \            |       \          /
     /      \       /      \           |        \        /
 +----+   +----+   +----+   +----+     |      +----+  +----+
 | 16 |   | 21 |   | 99 |   | 12 |     |      | 21 |  | 99 |
 +----+   +----+   +----+   +----+     |      +----+  +----+
          / \       \                  |       / \      \
         /   \       \                 |      /   \      \
      +----+ +----+   +----+           |  +----+ +----+   +----+
      | 45 | | 33 |   | 67 |           |  | 45 | | 33 |   | 67 |
      +----+ +----+   +----+           |  +----+ +----+   +----+
              \                        |
               \                       |
               +----+                  |
               | 22 |                  |
               +----+                  |
```

You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the tree class nor create any data structures such as arrays, lists, etc. You should not construct any new node objects or change the data of any nodes. For full credit, your solution must be recursive and properly utilize the x = change(x) pattern.

Assume that you are adding this method to the IntTree class as defined below:

```
public class IntTree {
   private IntTreeNode overallRoot;
   ...
}

public void evenLevels() {
   overallRoot = evenLevels(overallRoot, 1);
}

private IntTreeNode evenLevels(IntTreeNode node, int height) {
   if (node!= null) {

      node.left = evenLevels(node.left, height + 1);
      node.right = evenLevels(node.right, height + 1);

      if(node.right == null && node.left == null && height % 2 == 1){
         node = null;
      }
   }
   return node;
}
```

Chapter 18: Advanced Data Structures

Exercise 18.1: addAllHasIntSet

Write a method named addAll that could be placed inside the HashIntSet class. This method accepts another HashIntSet as a parameter and adds all elements from that set into the current set, if they are not already present. For example, if a set s1 contains [1, 2, 3] and another set s2 contains [1, 7, 3, 9], the call of s1.addAll(s2); would change s1 to store [1, 2, 3, 7, 9] in some order.

You are allowed to call methods on your set and/or the other set. Do not modify the set passed in. This method should run in O(*N*) time where *N* is the number of elements in the parameter set passed in.

```
public void addAll(HashIntSet other) {
   for (Node front : other.elementData) {
      Node current = front;
      while (current != null) {
         add(current.data);
         current = current.next;
      }
   }
}
```

Exercise 18.2:containsAllHashIntSet

Write a method in the HashIntSet class called containsAll that accepts another hash set as a parameter and returns true if your set contains every element from the other set. For example, if the set stores [-2, 3, 5, 6, 8] and the method is passed [3, 6, 8], your method would return true. If the method were passed [3, 6, 7, 8], your method would return false because your set does not contain the value 7.

```
public boolean containsAll(HashIntSet other) {
   for (Node front : other.elementData) {
      Node current = front;
      while (current != null) {
         if (!contains(current.data)) {
            return false;
         }
         current = current.next;
      }
   }
   return true;
}
```

Exercise 18.3: equalsHashIntSet
Write a method in the HashIntSet class called equals that accepts another object as a parameter and returns true if the object is another HashIntSet that contains exactly the same elements. The internal hash table size and ordering of the elements does not matter, only the sets of elements themselves.

```
public boolean equals(Object o) {
   if (!(o instanceof HashIntSet)) {
      return false;
   }
   HashIntSet other = (HashIntSet) o;
   for (Node front : elementData) {
      Node current = front;
      while (current != null) {
         if (!other.contains(current.data)) {
            return false;
         }
         current = current.next;
      }
   }
   return size == other.size();
}
```

Exercise 18.4: removeAllHashIntSet
Write a method in the HashIntSet class called removeAll that accepts another hash set as a parameter and ensures that this set does not contain any of the elements from the other set. For example, if the set stores [-2, 3, 5, 6, 8] and the method is passed [2, 3, 6, 8, 11], your set would store [-2, 5] after the call.

```
public void removeAll(HashIntSet other) {
   for (Node front : other.elementData) {
      Node current = front;
      while (current != null) {
         if (contains(current.data)) {
            remove(current.data);
         }
         current = current.next;
      }
   }
}
```

Exercise 18.5: retainAllHashIntSet
Write a method in the HashIntSet class called retainAll that accepts another hash set as a parameter and removes all elements from this set that are not contained in the other set. For

example, if the set stores [-2, 3, 5, 6, 8] and the method is passed [2, 3, 6, 8, 11], your set would store [3, 6, 8].

```
public void retainAll(HashIntSet other) {
   for (int i = 0; i < elementData.length; i++) {
      if (elementData[i] != null) {
         while (elementData[i] != null && !other.contains(elementData[i].data)) {
            elementData[i] = elementData[i].next;
            size--;
         }
         Node current = elementData[i];
         while (current != null && current.next != null) {
            if (!other.contains(current.next.data)) {
               current.next = current.next.next;
               size--;
            } else {
               current = current.next;
            }
         }
      }
   }
}
```

Exercise 18.6: toArrayHashIntSet
Write a method in the HashIntSet class called toArray that returns the elements of the set as a filled array. The order of the elements in the array is not important as long as all elements from the set are present in the array, with no extra empty slots before or afterward.

```
public int[] toArray() {
   int[] result = new int[size];
   int i = 0;
   for (Node front : elementData) {
      Node current = front;
      while (current != null) {
         result[i] = current.data;
         i++;
         current = current.next;
      }
   }
   return result;
}
```

Exercise 18.7: toStringHashIntSet
Write a method in the HashIntSet class called toString that returns a string representation of the elements in the set, such as "[-2, 3, 5, 6, 8]". The order of the elements in the string does not matter as long as they are all present in the proper format. Do not use any other auxiliary

collections to help you. Do not list any empty or meaningless indexes in the string. Do not modify the state of the set.

```
public String toString() {
   if (isEmpty()) {
      return "[]";
   } else {
      String result = "[";
      for (Node front : elementData) {
         Node current = front;
         while (current != null) {
            if (result.length() > 1) {
               result += ", ";
            }
            result += current.data;
            current = current.next;
         }
      }
      result += "]";
      return result;
   }
}
```

Exercise 18.8: descending
Write a method called descending that accepts an array of integers and rearranges the integers in the array to be in descending order using a PriorityQueue as a helper. For example, if the array passed stores [42, 9, 22, 17, -3, 81], after the call the array should store [81, 42, 22, 17, 9, -3].

```
public static void descending(int[] a) {
   Queue<Integer> pq = new PriorityQueue<Integer>(100, Collections.reverseOrder());
   for (int n : a) {
      pq.add(n);
   }
   for (int i = 0; i < a.length; i++) {
      a[i] = pq.remove();
   }
}
```

Exercise 18.9: kthSmallest
Write a method called kthSmallest that accepts a PriorityQueue of integers and an integer *k* as parameters and returns the *k*th-smallest integer from the priority queue (where *k*=1 would represent the very smallest). For example, if the queue passed stores the integers [42, 50, 45, 78, 61] and *k* is 4, return the fourth-smallest integer, which is 61. If *k* is 0 or negative or greater than the size of the queue, throw an IllegalArgumentException. If your method modifies the state of the queue during its computation, it should restore the queue before it returns. You may use one stack or queue as auxiliary storage.

```
public static int kthSmallest(PriorityQueue<Integer> pq, int k) {
    if (k <= 0 || k > pq.size()) {
        throw new IllegalArgumentException();
    }

    Queue<Integer> backup = new LinkedList<Integer>();
    int size = pq.size();
    int kth = 0;

    for (int i = 0; i < size; i++) {
        int n = pq.remove();
        if (i == k - 1) {
            kth = n;
        }
        backup.add(n);
    }

    while (!backup.isEmpty()) {
        pq.add(backup.remove());   // restore queue
    }

    return kth;
}
```

Exercise 18.10: isConsecutive

Write a method called isConsecutive that accepts a PriorityQueue of integers as a parameter and returns true if the queue contains a sequence of consecutive integers starting from the front of the queue. Consecutive integers are integers that come one after the other, as in 5, 6, 7, 8, 9, etc., so if the queue stores [7, 8, 9, 10, 11], your method should return true. (Also return true if passed an empty queue.) If your method modifies the state of the queue during its computation, it should restore the queue before it returns. You may use one stack or queue as auxiliary storage.

```
public static boolean isConsecutive(PriorityQueue<Integer> pq) {
    if (pq.size() <= 1) {
        return true;
    }

    Queue<Integer> backup = new LinkedList<Integer>();
    int prev = pq.remove();
    backup.add(prev);
    boolean consecutive = true;

    while (!pq.isEmpty()) {
        int next = pq.remove();
        if (prev + 1 != next) {
            consecutive = false;
        }
        backup.add(next);
        prev = next;
    }

    while (!backup.isEmpty()) {
        pq.add(backup.remove());   // restore queue
    }

    return consecutive;
}
```

Exercise 18.11: removeDuplicates
Write a method called removeDuplicates that accepts a PriorityQueue of integers as a parameter and modifies the queue's state so that any element that is equal to another element in the queue is removed. For example, if the queue stores [7, 7, 8, 8, 8, 10, 45, 45], your method should modify the queue to store [7, 8, 10, 45]. You may use one stack or queue as auxiliary storage.

```
public static void removeDuplicates(PriorityQueue<Integer> pq) {
    if (pq.size() <= 1) {
        return;
    }

    Queue<Integer> unique = new LinkedList<Integer>();
    int prev = pq.remove();
    unique.add(prev);

    while (!pq.isEmpty()) {
        int next = pq.remove();
        if (prev != next) {
            unique.add(next);
            prev = next;
        }
    }

    while (!unique.isEmpty()) {
        pq.add(unique.remove());   // restore queue
    }
}
```

Exercise 18.12: stutter
Write a method called stutter that accepts a PriorityQueue of integers as a parameter and replaces every value in the queue with two occurrences of that value. For example, if the queue stores [7,

8, 10, 45], your method should modify the queue to store [7, 7, 8, 8, 10, 10, 45, 45]. You may use one stack or queue as auxiliary storage.

```
public static void stutter(PriorityQueue<Integer> pq) {
   Queue<Integer> temp = new LinkedList<Integer>();
   while (!pq.isEmpty()) {
      temp.add(pq.remove());
   }

   while (!temp.isEmpty()) {
      int n = temp.remove();
      pq.add(n);
      pq.add(n);
   }
}
```

Exercise 18.13: toArrayHeapIntPriorityQueue
Write a method in the HeapIntPriorityQueue class called toArray that returns the elements of the queue as a filled array. The order of the elements in the array is not important as long as all elements from the queue are present in the array, with no extra empty slots before or afterward.

```
public int[] toArray() {
   int[] result = new int[size];
   for (int i = 0; i < size; i++) {
      result[i] = elementData[i + 1];
   }
   return result;
}
```

Exercise 18.14: toStirngHeapIntPriorityQueue
Write a method in the HeapIntPriorityQueue class called toString that returns a string representation of the elements in the queue, such as "[42, 50, 45, 78, 61]". The order of the elements in the string does not matter as long as they are all present in the proper format.

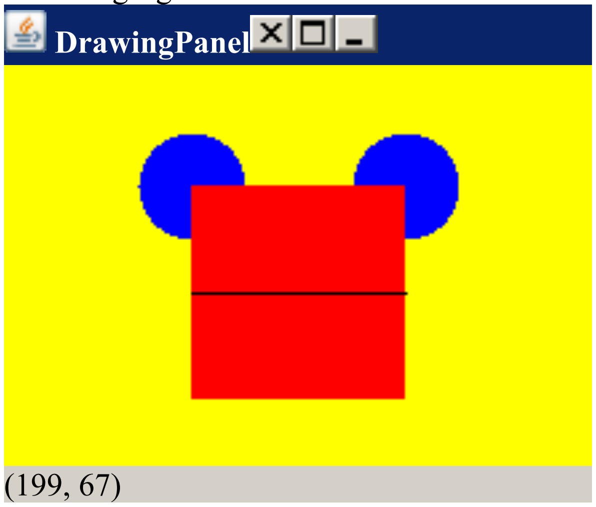
```
public String toString() {
   if (isEmpty()) {
      return "[]";
   } else {
      String result = "[" + elementData[1];
      for (int i = 2; i <= size; i++) {
         result += ", " + elementData[i];
      }
      result += "]";
      return result;
   }
}
```

Exercise 18.15: mergeHeapIntPriorityQueue
Write a method in the HeapIntPriorityQueue class called merge that accepts another HeapIntPriorityQueue as a parameter and adds all elements from the other queue into the current queue, maintaining proper heap order such that the elements will still come out in ascending order when they are removed. Your code should not modify the queue passed in as a parameter. (Recall that objects of the same class can access each other's private fields.)

```
public void merge(HeapIntPriorityQueue other) {
   for (int i = 1; i <= other.size; i++) {
      add(other.elementData[i]);
   }
}
```

Supplement 3G: Graphics

Exercise 3G.1: MickeyBox
Write a complete program in a class named MickeyBox that uses the DrawingPanel to draw the following figure:



The window is 220 pixels wide and 150 pixels tall. The background is yellow. There are two blue ovals of size 40 x 40 pixels. The left oval's top-left corner is located at position (50, 25), and the two ovals' top-left corners are 80 pixels apart horizontally. There is a red square whose top two corners exactly intersect the centers of the two ovals. Lastly, there is a black horizontal line through the center of the square.
(You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

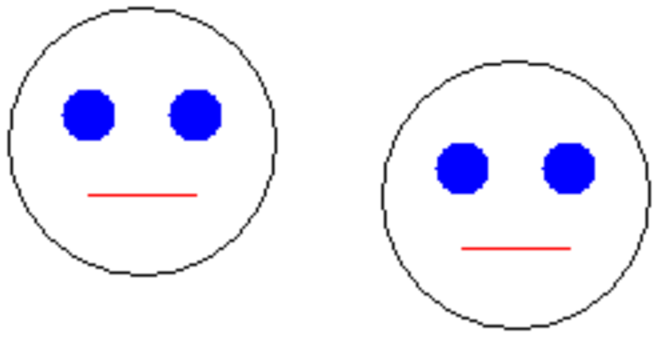
```
public class MickeyBox {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(220, 150);
        Graphics g = panel.getGraphics();
        g.setColor(Color.YELLOW);
        g.fillRect(0, 0, 220, 150);
        g.setColor(Color.BLUE);
        g.fillOval(50, 25, 40, 40);
        g.fillOval(130, 25, 40, 40);
        g.setColor(Color.RED);
        g.fillRect(70, 45, 80, 80);
        g.setColor(Color.BLACK);
        g.drawLine(70, 85, 150, 85);
    }
}
```

Exercise 3G.2: MickeyBox
Modify your MickeyBox program from the previous exercise into a new class MickeyBox2 so that the figure is drawn by a method called drawFigure. The method should accept three

parameters: the Graphics g of the DrawingPanel on which to draw, and two ints specifying the (x, y) position of the top-left corner of the figure. Use the following heading for your method
public static void **drawFigure**(Graphics g, int x, int y)
Set your DrawingPanel's size to 450 x 150 pixels, and use your drawFigure method to place two figures on it. One figure should be at position (50, 25) and the other should be at position (250, 45).

```
public class MickeyBox2 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(450, 150);
        Graphics g = panel.getGraphics();
        g.setColor(Color.YELLOW);
        g.fillRect(0, 0, 450, 150);
        drawFigure(g, 50, 25);
        drawFigure(g, 250, 45);
    }

    public static void drawFigure(Graphics g, int x, int y) {
        g.setColor(Color.BLUE);
        g.fillOval(x, y, 40, 40);
        g.fillOval(x + 80, y, 40, 40);
        g.setColor(Color.RED);
        g.fillRect(x + 20, y + 20, 80, 80);
        g.setColor(Color.BLACK);
        g.drawLine(x + 20, y + 60, x + 100, y + 60);
    }
}
```

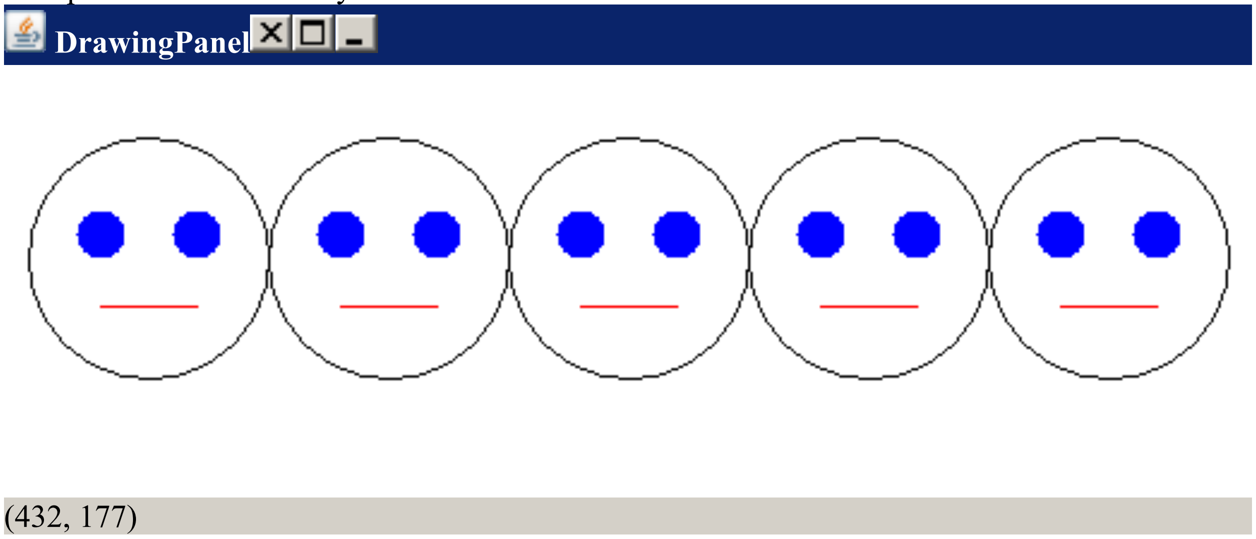
Exercise 3G.3: Face
Suppose you have the following existing program called Face that uses the DrawingPanel to draw a face figure. Modify the program to draw the graphical output shown below. Do so by writing a parameterized static method that draws a face at different positions. The window size should be changed to 320 x 180 pixels, and the two faces' top-left corners are at (10, 30) and (150, 50).



(256, 165)
(You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

```
public class Face {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(320, 180);
        Graphics g = panel.getGraphics();
        drawFace(g, 10, 30);
        drawFace(g, 150, 50);
    }

    public static void drawFace(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.drawOval(x, y, 100, 100);   // face outline

        g.setColor(Color.BLUE);
        g.fillOval(x + 20, y + 30, 20, 20);     // eyes
        g.fillOval(x + 60, y + 30, 20, 20);

        g.setColor(Color.RED);          // mouth
        g.drawLine(x + 30, y + 70, x + 70, y + 70);
    }
}
```

Exercise 3G.4: Face2
Modify your Face program from the previous exercise into a new class Face2 to draw the new output shown below. The window size should be changed to 520 x 180 pixels, and the faces' top-left corners are at (10, 30), (110, 30), (210, 30), (310, 30), and (410, 30). Draw the figures using a loop to avoid redundancy.

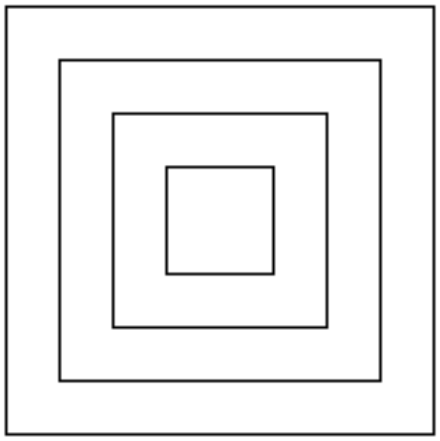


(432, 177)

(You don't need to include any import statements at the top of your program.)

```
public class Face2 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(520, 180);
        Graphics g = panel.getGraphics();
        for(int i = 0; i < 5; i++)
            drawFace(g, 10 + 100 * i, 30);
    }
    public static void drawFace(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.drawOval(x, y, 100, 100);     // face outline

        g.setColor(Color.BLUE);
        g.fillOval(x + 20, y + 30, 20, 20);     // eyes
        g.fillOval(x + 60, y + 30, 20, 20);

        g.setColor(Color.RED);          // mouth
        g.drawLine(x + 30, y + 70, x + 70, y + 70);
    }
}
```

Exercise 3G.5: ShowDesign
Write a complete program in a class named ShowDesign that uses the DrawingPanel to draw the following figure:



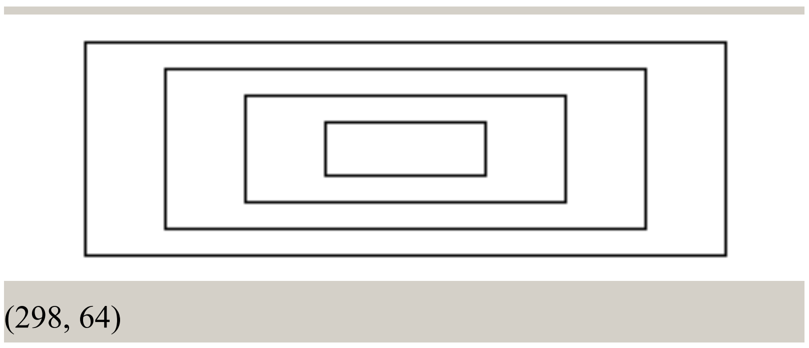
(161, 192)

The window is 200 pixels wide and 200 pixels tall. The background is white and the foreground is black. There are 20 pixels between each of the four rectangles, and the rectangles are concentric (their centers are at the same point). Use a loop to draw the repeated rectangles. (You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

```
public class ShowDesign {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(200, 200);
      Graphics g = panel.getGraphics();
      for(int i = 0; i < 4; i++)
         g.drawRect(20 + 20 * i, 20 + 20 * i, 160 - 40 * i, 160 - 40 * i);
   }
}
```

Exercise 3G.6: ShowDesign2

Modify your ShowDesign class from the previous exercise into a new class ShowDesign2 that has a method named showDesign that accepts parameters for the window width and height and displays the rectangles at the appropriate sizes. For example, if your showDesign method was called with values of 300 and 100, the window would look like the following figure.



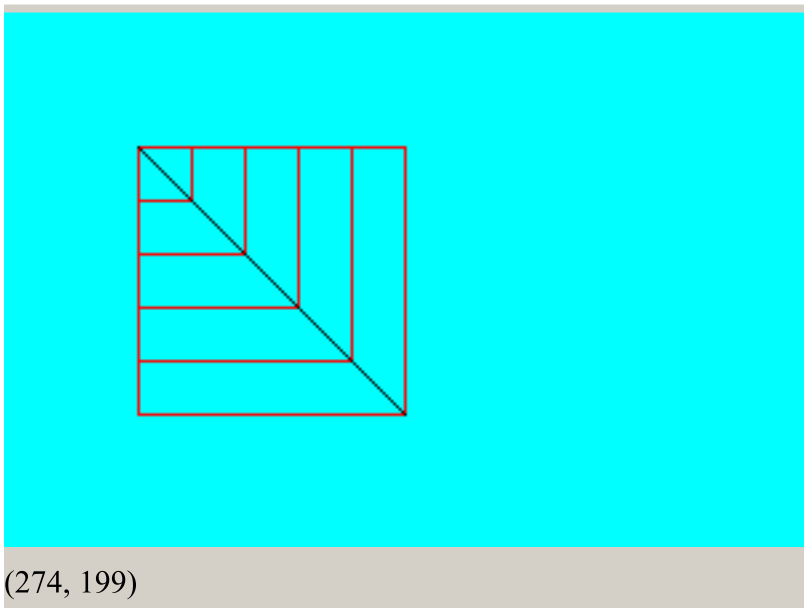
(298, 64)

(You don't need to include any import statements at the top of your program.)

```
public class ShowDesign2 {
   public static void main(String[] args) {
      showDesign(300, 100);
   }

   public static void showDesign(int width, int height) {
      DrawingPanel panel = new DrawingPanel(width, height);
      Graphics g = panel.getGraphics();
      for(int i = 1; i <= 4; i++)
         g.drawRect(30 * i, 10 * i, width - 60 * i, height - 20 * i);
   }
}
```

Exercise 3G.7 Squares

Write a program in a class named Squares that uses the DrawingPanel to draw the following figure:



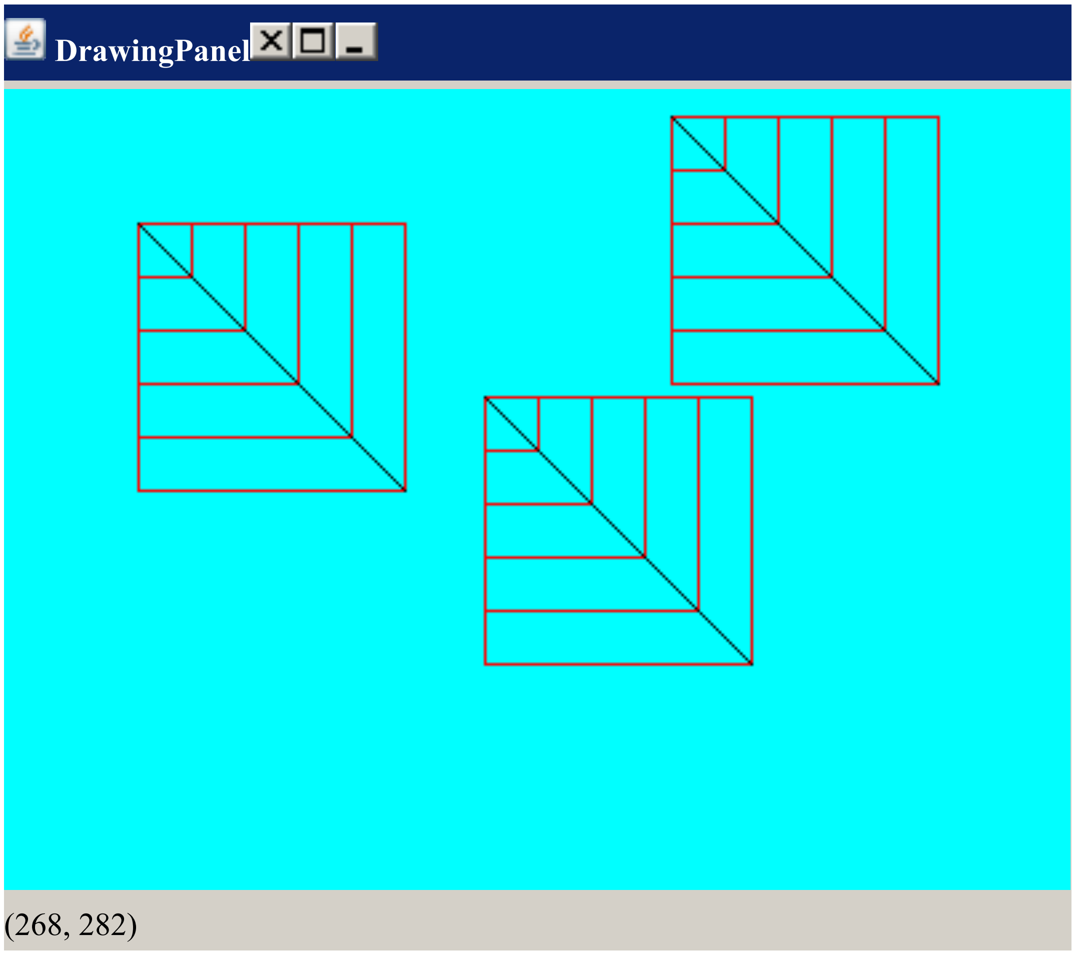
The drawing panel is 300 pixels wide by 200 pixels high. Its background is cyan. The horizontal and vertical lines are drawn in red and the diagonal line is drawn in black. The upper-left corner of the diagonal line is at (50,50). Successive horizontal and vertical lines are spaced 20 pixels apart. The diagonal line is drawn on top of the horizontal and vertical lines.

(You don't need to include any import statements at the top of your program.) (The next two exercises are modified versions of this program, so you can use the code you write here as a starting point for those exercises.)

```
public class Squares {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(300, 200);
      Graphics g = panel.getGraphics();
      g.setColor(Color.CYAN);
      g.fillRect(0, 0, 300, 200);
      g.setColor(Color.RED);

      for(int i = 1; i <= 5; i++)
         g.drawRect(50, 50, 20 * i, 20 * i);

      g.setColor(Color.BLACK);
      g.drawLine(50, 50, 150, 150);
   }
}
```

Exercise 3G.8: Sqaures2

Modify your Squares program from the previous exercise into a new class Squares2 that draws the following figures. (Go back to that problem and copy/paste your code here as a starting point.)



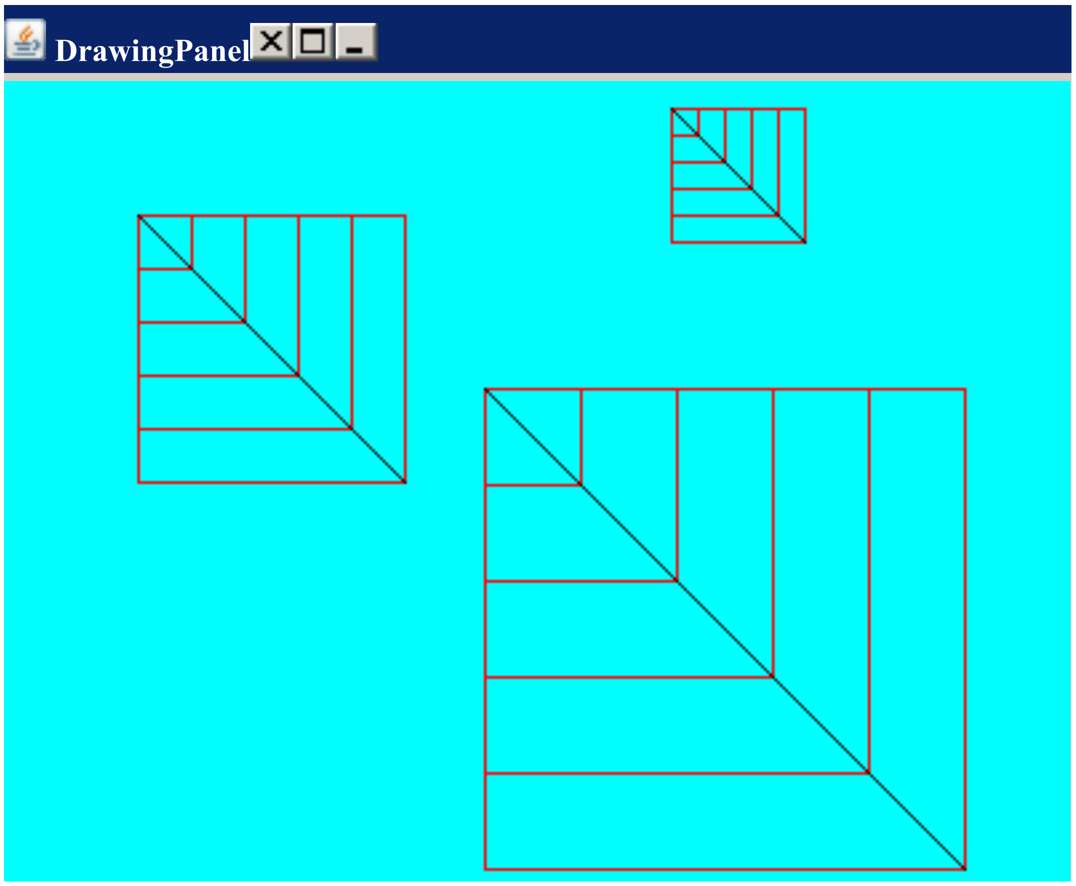
The drawing panel is now 400 by 300 pixels in size. The first figure is at the same position, (50,50). The other figures are at positions (250, 10) and (180, 115), respectively. Use one or more parameterized static methods to reduce the redundancy of your solution.

```
public class Squares2 {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(400, 300);
      Graphics g = panel.getGraphics();
      g.setColor(Color.CYAN);
      g.fillRect(0, 0, 400, 300);
      drawFigure(g, 50, 50);
      drawFigure(g, 250, 10);
      drawFigure(g, 180, 115);
   }
```

```
    public static void drawFigure(Graphics g, int x, int y) {
        g.setColor(Color.RED);

        for(int i = 1; i <= 5; i++)
            g.drawRect(x, y, 20 * i, 20 * i);

        g.setColor(Color.BLACK);
        g.drawLine(x, y, x + 100, y + 100);
    }
}
```

Exercise 3G.8: Sqaures2

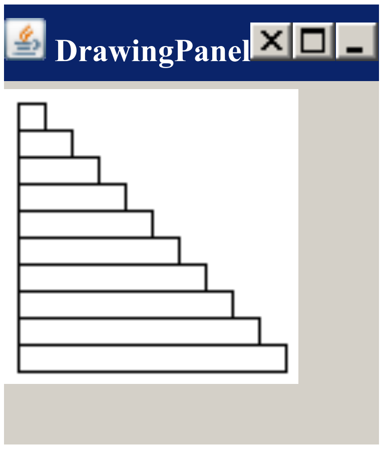
Modify your Squares2 program from the previous exercise into a new class Squares3 that draws the following figures. (Go back to that problem and copy/paste your code here as a starting point.) Parameterize your program so that the figures have the sizes shown below. The top-right figure has size 50, and the bottom-right figure has size 180.



```
public class Squares3 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(400, 300);
        Graphics g = panel.getGraphics();
        g.setColor(Color.CYAN);
        g.fillRect(0, 0, 400, 300);
        drawFigure(g, 50, 50, 100);
        drawFigure(g, 250, 10, 50);
        drawFigure(g, 180, 115, 180);
    }

    public static void drawFigure(Graphics g, int x, int y, int size) {
        int separation = size / 5;
        g.setColor(Color.RED);

        for(int i = 1; i <= 5; i++)
            g.drawRect(x, y, separation * i, separation * i);

        g.setColor(Color.BLACK);
        g.drawLine(x, y, x + size, y + size);
    }
}
```

Exerciser 3G.10: Stairs

Finish the given program called Stairs that uses the DrawingPanel to draw the figure shown below. The window is 110 x 110 px in size. The first stair's top-left corner is at position (5, 5). The first stair is 10 x 10 pixels in size. Each stair is 10 pixels wider than the one above it. *(If you're having trouble matching the output, make a table with the (x, y) coordinates and (width x height) sizes of the first five stairs. Note which values change and which ones stay the same.)*

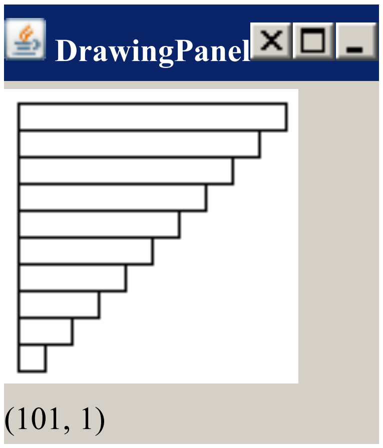


(You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

```
public class Stairs {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(110, 110);
      Graphics g = panel.getGraphics();
      for(int i = 0; i < 10; i++) {
         g.drawRect(5, 5 + 10 * i, 10 + 10 * i, 10);
      }
   }
}
```

Exerciser 3G.11a: Stairs2

Modify your Stairs program from the previous exercise to make a new class Stairs2 that draws the output shown below. Modify only the body of your loop. *(You may want to make a new table to find the expressions for x, y, width, and height for each new output.)*

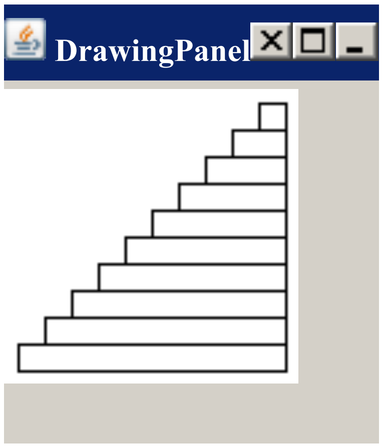


(You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

```
public class Stairs2 {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(110, 110);
      Graphics g = panel.getGraphics();
      for(int i = 0; i < 10; i++) {
         g.drawRect(5, 5 + 10 * i, 100 - 10 * i, 10);
      }
   }
}
```

Exerciser 3G.11b: Stairs3

Modify your Stairs program from the previous exercise to make a new class Stairs3 that draws the output shown below. Modify only the body of your loop. *(You may want to make a new table to find the expressions for x, y, width, and height for each new output.)*

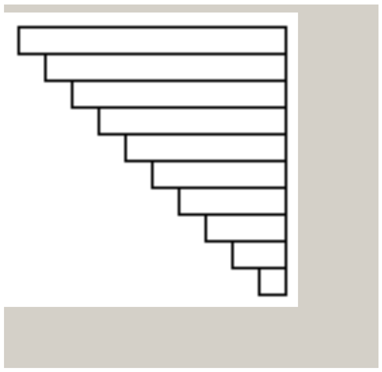


(You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

```
public class Stairs3 {
    public static void main(String[] args) {
        DrawingPanel panel = new DrawingPanel(110, 110);
        Graphics g = panel.getGraphics();
        for(int i = 0; i < 10; i++) {
            g.drawRect(95 - 10 * i, 5 + 10 * i, 10 + 10 * i, 10);
        }
    }
}
```

Exerciser 3G.11c: Stairs4

Modify your Stairs program from the previous exercise to make a new class Stairs4 that draws the output shown below. Modify only the body of your loop. *(You may want to make a new table to find the expressions for x, y, width, and height for each new output.)*

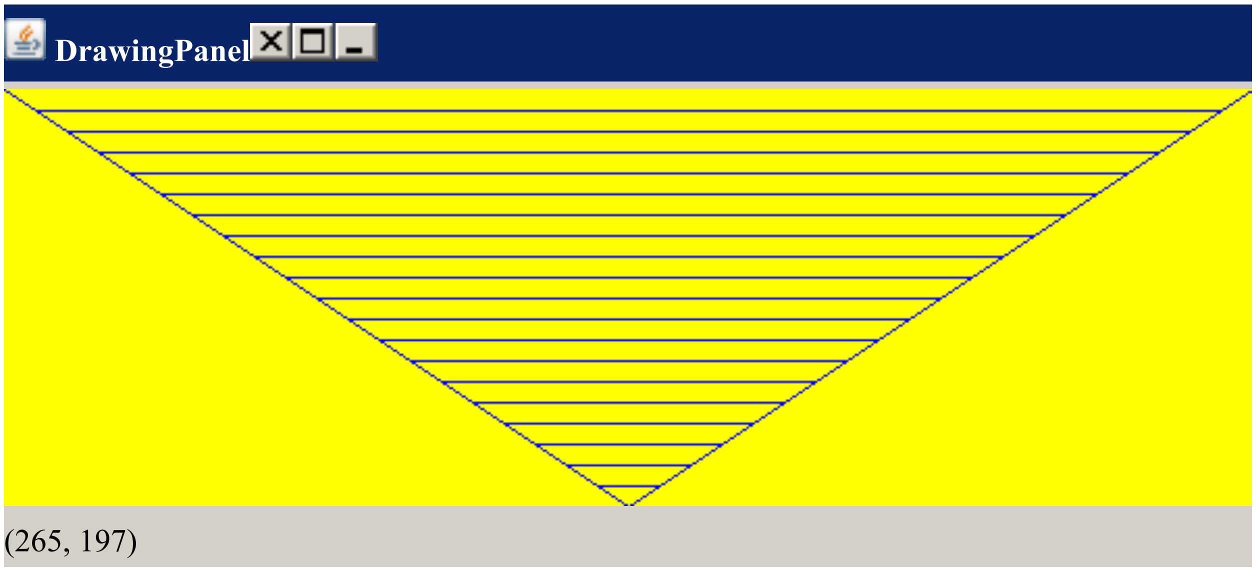


(You don't need to include any import statements at the top of your program.) (The next exercise is a modified version of this program, so you can use the code you write here as a starting point for that exercise.)

```
public class Stairs4 {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(110, 110);
      Graphics g = panel.getGraphics();
      for(int i = 0; i < 10; i++) {
         g.drawRect(5 + 10 * i, 5 + 10 * i, 100 - 10 * i, 10);
      }
   }
}
```

Exercise 3G.12: Triangle

Using the DrawingPanel class, write a Java class named Triangle that produces the following figure:



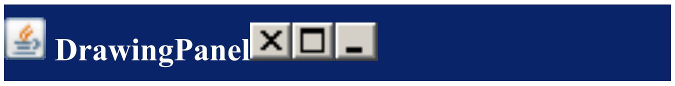
- size: 600x200
- background color: yellow
- line color: blue
- vertical spacing between lines: 10 px

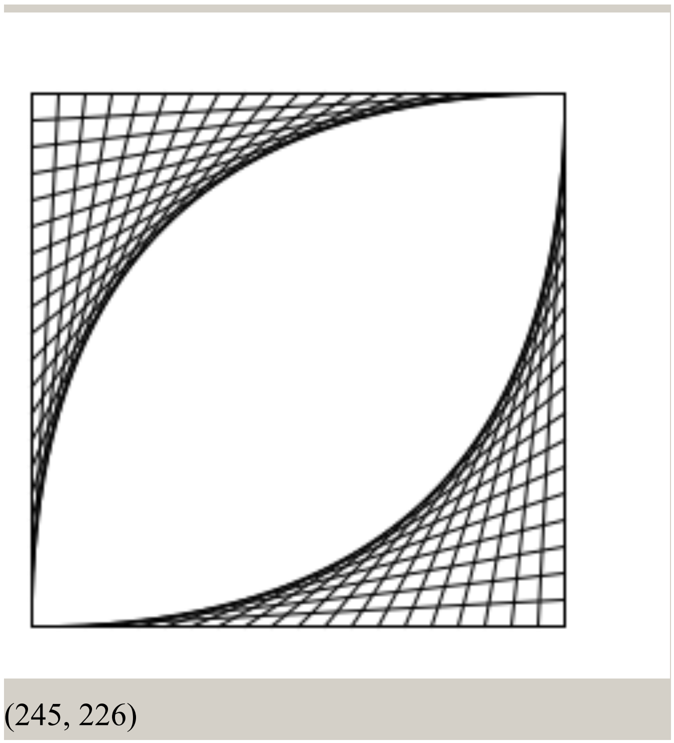
The diagonal lines connect at the bottom in the middle.

```
public class Triangle {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(600, 200);
      Graphics g = panel.getGraphics();
      g.setColor(Color.YELLOW);
      g.fillRect(0, 0, 600, 200);
      g.setColor(Color.BLUE);

      for(int y = 10; y < 200; y += 10) {
         int x1 = 3 * y / 2;
         int x2 = 3 * (400 - y) / 2;
         g.drawLine(x1, y, x2, y);
      }

      g.drawLine(0, 0, 300, 200);
      g.drawLine(300, 200, 600, 0);
   }
}
```

Exercise 3G.13: Football

Using the DrawingPanel class, write a Java class named Football that produces the following figure:




(245, 226)

Though the figure looks to contain curves, it is made entirely of straight lines. The window is 250 x 250 pixels in size, and there is an outer rectangle from (10, 30) to (210, 230). A set of black lines are drawn around the edges every 10 pixels. For example, along the top-left, there is a line from (10, 220) to (20, 30), a line from (10, 210) to (30, 30), a line from (10, 200) to (40, 30), ... and so on. Along the bottom-right, there is a line from (20, 230) to (210, 220), a line from (30, 230) to (210, 210), and so on.

```
public class Football {
   public static void main(String[] args) {
      DrawingPanel panel = new DrawingPanel(250, 250);
      Graphics g = panel.getGraphics();
      g.setColor(Color.black);
                              g.drawRect(10, 30, 200, 200);
                              for(int i = 0; i < 20; i++) {
         int x1 = 10;
                                    int y1 = 220 - 10 * i;
                                         int x2 = 20 + 10 * i;
                                         int y2 = 30;
                                 g.drawLine(x1, y1, x2, y2);
                                 g.drawLine(x2, 230, 210, y1);
      }
   }
}
```