Advanced Placement in Computer Science

Leon Tabak

19 July 2018

Contents

1 Logic **7**

1.1 Some symbols have more than one meaning. 20

2 Why Java? **23**

3 Parts of a Java Program **25**

3.1 Source code: Simple.java 25

3.2 Source code: SimpleAnnotated.java 25

4 Variables **29**

4.1 The rules are different. 29

4.2 Source code: Asia.java 30

4.3 Primitive types and reference types 30

4.4 Source code: ParameterPassing.java 30

5 Methods **33**

5.1 Source code: TriangularNumbers.java 33

6 Practice with methods: computing means **37**

6.1 Source code: Means.java 37

7 Practice with methods: computing distances **39**

7.1 Source code: Distances.java 39

8 Practice with methods: weighted averages **41**

8.1 Source code: BezierFrame.java 41
8.2 Source code: BezierPanel.java 42

9 My favorite definition of computer science 49
9.1 A first look at complexity: bank accounts 49
9.1.1 Source code: NumberedAccounts.java 50
9.1.2 Sample input 51
9.1.3 What to do next? 51
9.1.4 Source code: NamedAccounts.java 51
9.1.5 Sample input 52

10 Reading and writing files. 53
10.1 Source code: FileIO.java 53

11 Traces of sorts 55
11.1 Trace of selection sort 56
11.2 Trace of insertion sort 57
11.3 Trace of merge sort 57

12 Selection sort 59
12.1 Source code: RandomNumbers.java 59
12.2 Source code for exercise: Sorts.java 61
12.3 Source code for solution to exercise: SortsSolution.java 69

13 Classes as blueprints 81
13.1 Arithmetic with fractions 81
13.1.1 Greatest common divisor 81
13.1.2 Addition 82
13.1.3 Subtraction 82
13.1.4 Multiplication 83
13.1.5 Division 83

14 A Java class that models a fraction 85

14.1 Source code: Fraction.java 85

15 Sorting Fractions 89

15.1 Source code: ComparableFraction.java 89

16 Review 93

17 What we have learned and what comes next 121

Lesson 1

Logic

Our work will be easier if we all speak the same language. It will be easier if we can find a language that gives us the means to express ourselves concisely.

---

1. I say: "I ran today." This sentence is either true or false. Let's call this sentence **A**.

   I say: "I swam today." This sentence is also true or false. Let's call this sentence **B**.

   I can combine **A** and **B** to make a bigger sentence: "I ran today and I swam today." This kind of combination is called *conjunction*. *Conjunction* means *and*. Again, this sentence is either true or false.

   When is the conjunction of **A** and **B** true? When is it false?

   Let's answer these questions in a table. There are four possible assignments of true/false (T/F) values to **A** and **B**. This gives us a table with four rows:

A	**B**	**A and B**
T	T	
T	F	
F	T	
F	F	

   Complete the table by filling in the right-most column.

Answer:

A	**B**	**A and B**
T	T	T
T	F	F
F	T	F
F	F	F

2. I can also combine **A** and **B** with disjunction. *Disjunction* means *or*.

   "I ran today or I swam today."

   There is more than one way in which we might interpret this sentence. In this case, let's allow for the possibility of two kinds of exercise on the same day.

   When is the disjunction of **A** and **B** true? When is it false?

   Here is the beginnings of the table:

A	**B**	**A or B**
T	T	
T	F	
F	T	
F	F	

   Complete the table by filling in the right-most column.

Answer:

A	**B**	**A or B**
T	T	T
T	F	T
F	T	T
F	F	F

3. The word "or" can be inclusive or exclusive.

   "Inclusive" means that both halves of a statement could be true. I might run and swim. I can do both.

   "Exclusive" means that only one half of a statement can be true. When I say that my new friend's birthday is in September or October, I mean either September or October.

Here's another example: Imagine that you are writing a paper and need to know the name of the capital city of South Dakota. You call out to your roommate.

"What's the capital of South Dakota?"

Your roommate answers. "I am sure that it is Pierre or Bismark."

South Dakota has only one capital city. Your roommate means that the capital is either Pierre or Bismark.

Let **A** be the sentence: "Pierre is the capital of South Dakota."

Let **B** be the sentence: "Bismark is the capital of South Dakota."

Let **XOR** mean "exclusive or."

A XOR B means "Either **A** or **B**."

There are four possibilities:

- Your roommate's statement is false in the case that neither Pierre or Bismark is the capital of South Dakota.
- Your roommate's statement is false in the case that both Pierre and Bismark are capitals of South Dakota.
- Your roommate's statement is true in the case that Pierre is the capital and Bismark is not the capital.
- Your roommate's statement is true in the case that Bismark is the capital and Pierre is not the capital.

Each of these possibilities can be described in one row of a truth table. Write the truth table that summarizes these four possibilities.

A	**B**	**A XOR B**
T	T	
T	F	
F	T	
F	F	

Answer:

A	**B**	**A XOR B**
T	T	F
T	F	T
F	T	T
F	F	F

4. On Tuesday afternoon, I say: "If the sun shines this afternoon, then I will run."

   This sentence contains two smaller sentences. One sentence is "The sun shines this afternoon." The other sentence is "I will run." Let's use **A** to refer to the first sentence and **B** to refer to the second sentence.

   A sentence with this form is an *implication*: If **A** then **B**.

   Here's the truth table for the implication function.

A	**B**	**A** $\to$ **B**
T	T	T
T	F	F
F	T	T
F	F	T

   On Wednesday you ask me: "Did the sun shine?" and "Did you run?"

   With the answers to these questions you will be able to determine whether or not I fibbed on Tuesday.

   There is only one case that makes Tuesday's statement a fib. What is it?

   Answer:

   If the sun did shine, but I did not run, then I have fibbed.

   If the sun did not shine, no matter whether I run or not, I am innocent of any fibbing.

   If the sun did shine and I did run, then I kept my promise and there is no fib.

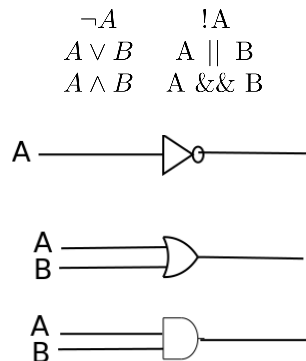
5. Computer scientists, electrical engineers, and mathematicians all prefer different symbols for conjuncation, disjunction, and logical negation.

   This is an example of a common challenge that arises in our learning.

   Find and identify symbols used by computer scientists, electrical engineers, and mathematicians to represent. . .

   (a) conjunction
   (b) disjunction
   (c) logical negation

   Here's a start.



Answer:

negation	$\neg A$	!A
disjunction	$A \vee B$	A || B
conjunction	$A \wedge B$	A && B

The image at the top of the figure (a triangle with a small circle) represents negation.

The image in the middle of the figure (a shape bounded by one concave curve and two convex curves) represents disjunction.

The image at the bottom of the figure (a rectangle with one edge replaced by a half-circle) represents conjunction.

6. Universities are divided into colleges. Colleges are divided into departments. Departments offer courses. These divisions might suggest boundaries between different fields of study; in fact, there are overlaps among many subjects.

   Libraries identify books using the Dewey Decimal Classification or the Library of Congress Classification. The Russell D. Cole Library at Cornell College orders books with the Dewey Decimal system. One might hope to find all books about computer science on one shelf, but in fact you can find books about computer science in several parts of the library's collection:

   For example,. . .

   - The call number for *The Structure and Interpretation of Computer Programs*, by Harold Abelson, Gerald Jay Sussman, and Julie Sussman, is 005.133 Ab34st 1996.

- The call number for *Fractals: Form, Chance, and Dimension*, by Benoit Mandelbrot, is 516 M312f.
- The call number for *The Algorithmic Beauty of Plants*, by Przemyslaw Prusinkiewicz and Astrid Lindenmayer, is 581.31 P953al.
- The call number for *Broken Genius: the rise and fall of William Shockley, creator of the electronic age*, by Joel N. Shurkin, is 621.3810924 Sh72s 2006.

These books are shelved among, respectively, books about knowledge and systems of knowledge (not far from philosophy), mathematics, biology, and technology.

Similarly, the study of logic draws from the study of other subjects. We can also apply our knowledge of logic in the study of many other subjects. In particular, there is an overlap between the study of logic and the study of sets.

- The elements of the union of the sets **A** and **B** are all elements that are members of **A** or **B**.
- The elements of the intersection of the sets **A** and **B** are all elements that are members of **A** and **B**.

(a) With which of these two symbols do mathematicians denote union?

(b) With which of these two symbols do mathematicians denote intersection?

(c) Which symbol most closely resembles the symbol for conjunction (and)?

(d) Which symbol most closely resembles the symbol for disjunction (or)?

$$\cap$$
$$\cup$$

Answer:

(a) $\cup$ denotes a union

(b) $\cap$ denotes an intersection

(c) $\cup$ resembles $\vee$ (union is related to disjunction)

(d) $\cap$ resembles $\wedge$ (intersection is related to conjunction)

7. The truth table for a Boolean function of one variable contains two rows and two columns. There are exactly four different ways to fill the right-most column, so there are four Boolean functions of a single variable:

   These functions are the...

   - false function
   - identify function
   - logical negation function (also called the not function)
   - true function

   Label the following truth tables with the name of the corresponding function.

   (a)

A	f(**A**)
T	T
F	T

   (b)

A	f(**A**)
T	F
F	F

   (c)

A	f(**A**)
T	F
F	T

   (d)

A	f(**A**)
T	T
F	F

Answer:

(a) **true function**

A	f(**A**)
T	T
F	T

(b) **false function**

A	f(**A**)
T	F
F	F

(c) **not function**

A	f(**A**)
T	F
F	T

(d) **identity function**

A	f(**A**)
T	T
F	F

8. A Boolean function of two variables has four rows and three columns.

The truth table for each Boolean function of two variables looks like this. . .

A	**B**	**f(A,B)**
T	T	?
T	F	?
F	T	?
F	F	?

How many different Boolean functions of two variables are there?

The answer to this question is the same as the answer to the question: In how many different ways can the right-most column of this table be filled with T's and F's?

Answer:

There are 16 different Boolean functions of two variables.

Here are the 16 ways in which the right-most column of the truth table can be filled:

T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F

9. NAND and NOR are two special Boolean functions of two variables. What makes them special?

Answer:

It is possible to construct any Boolean function using just NAND functions.

It is also possible to construct any Boolean function using just NOR functions.

A computer engineer needs only one kind of component to build any digital circuit. In particular, it is possible to build all of the arithmetic and control circuits of a computer using just NAND gates or just NOR gates.

10. What is the opposite (the logical negation) of "I will pack my jacket and my umbrella."

    Let **A** be the sentence: "I will pack my jacket."

    Let **B** be the sentence: "I will pack my umbrella."

    Then the truth table looks like this:

A	**B**	$\mathbf{A} \wedge \mathbf{B}$
T	T	T
T	F	F
F	T	F
F	F	F

I told you the truth when I said "I will pack my jacket and my umbrella" only if both the jacket and the umbrella are now in my bag.

To negate the statement, change every T in the right-most column to an F and every F to a T:

A	**B**	$\mathbf{A} \wedge \mathbf{B}$	$\neg\,(\mathbf{A} \wedge \mathbf{B})$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

The right-most column in this table specifies another Boolean function. **NAND** means **N**ot **AND**.

AND is **true** only when its two inputs are **true**.

NAND is **true** except when its two inputs are **true**.

Write the logical negation of "I will pack my jacket and my umbrella" in English. Try to write a sentence that sounds natural, and not a sentence that sounds like the answer to a quiz on mathematics.

Answer:

"I will not pack my jacket, or I will not pack my umbrella, or I will pack neither my jacket nor my umbrella."

Equivalently,. . .

"I will leave behind my jacket or my umbrella (or both)."

11. **A NAND B** is a function of two Boolean variables, but **A NAND A** is a function of just one Boolean variable.

    What is the name of the function?

    (Hint: It might help to write the truth table for **A NAND B** and then note that the truth table for **A NAND A** contains two rows borrowed from the truth table for **A NAND B**.)

Answer:

A NAND A is **true** when **A** is **false** and **false** when **A** is **true**.

A NAND A is the **NOT** (logical negation) function.

12. Not only are there Boolean functions of one variable and Boolean functions of two variables, there are Boolean functions of three, four, five, and any number of variables.

    How many rows are in the truth table for a Boolean function of n variables?

    An inductive approach works well here. Induction means beginning with small cases, looking for a pattern, and developing a rule that applies for all cases.

    You have seen that there are two rows in the truth table for a Boolean function of one variable.

    You have seen that there are four rows in the truth table for a Boolean function of two variables.

    Let's think about the truth table for a Boolean function of three variables. We have two choices for the value of the first variable, two choices for the value of the second variable, and two choices for the value of the third

variable. Because these are independent choices, the number of distinct ways of composing a row is a product: $2 \times 2 \times 2 = 8$.

This is the form of a table for a Boolean function of three variables. The values that correspond to each possible combination of inputs are "tbd" (to-be-determined).

A	**B**	**C**	f(**A**, **B**, **C**)
T	T	T	tbd
T	T	F	tbd
T	F	T	tbd
T	F	F	tbd
F	T	T	tbd
F	T	F	tbd
F	F	T	tbd
F	F	F	tbd

Here is a summary of what we know:

Number of variables	Number of rows in truth table
1	2
2	4
3	8

How many rows are in the truth table for a Boolean function of n variables?

Answer:

There are 2^n rows in the truth table for a Boolean function of n variables.

It follows that there are 2^{2^n} Boolean functions of n variables.

13. We will need an understanding of logic in order to understand how computer engineers design circuits that combine data arithmetically.

    Information is a measure of our ability to distinguish among alternatives. The fewest number of alternatives from which we might choose is two, and so the smallest amount of information is the amount that specifies one of two possibilities. We might call these possibilities **true** and **false**, T and F, 0 and 1, or (if we are electrical engineers) we might use $0V$ and $+5V$.

In any case, the smallest amount of information is a bit—a "BInary Digit" is 0 or 1.

The use of binary numbers in the design of computers and computer programs is a convenience. Binary arithmetic is not an essential features of computers—we could build computers that store numbers in decimal form, but engineers can more easily design circuits that distinguish between two alternatives than they can design circuits that distinguish among ten alternatives.

Just as we can represent any number using only the digits 0–9, we can represent any number using only the digits 0–1.

The key is the use of place values.

You know that 2018 means two thousand and eighteen. You learned in grade school that the 8 means 8×1, the 1 means 1×10, and 0 means 0×100, and the 2 means 2×1000.

$$2018 = 2 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 8 \cdot 10^0$$

In a decimal number, each digit denotes a multiple of some power of ten.

In the same fashion, each digit in a binary number denotes a multiple of some power of two.

The binary number 1110 has the same value as the decimal number 14 because...

$$\begin{aligned} 1110_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 8 + 4 + 2 + 0 \\ & 14_{10} \end{aligned}$$

Now, let's think about how we can compute the sum of two one digit binary numbers.

A	**B**	**A + B**
1	1	10
1	0	01
0	1	01
0	0	00

If let 1 mean T and let 0 mean F, then this almost looks like a truth table. The problem lies in the right-most column. We expect just one 0 or 1 (not two) in each cell of a truth table.

Let "msb" mean "most significant bit" and let "lsb" mean "least signficant bit."

Here's a table that comes closer to what we have learned to expect. In fact, this table is two truth tables in one.

A	**B**	msb(**A** + **B**)	lsb(**A** + **B**)
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

It might be clearer to you if I split this table into its two parts. Here are the two truth tables:

A	**B**	msb(**A** + **B**)
1	1	1
1	0	0
0	1	0
0	0	0

A	**B**	lsb(**A** + **B**)
1	1	0
1	0	1
0	1	1
0	0	0

Now, imagine that you are no longer a software engineer. You are now a computer engineer. Your job is not to produce Python and Java code. Your job is to design digital logic circuits.

You have in front of you electronic components that compute various Boolean functions.

(a) Which Boolean function do you need to compute the least signficant bit in the sum of two bits?

(b) Which Boolean function do you need to compute the most signficant bit (this is also the carry bit when the addition of two bits is just part of the calculation of the sum of larger numbers)?

Answer:

(a) **XOR**

(b) **AND**

1.1 Some symbols have more than one meaning.

The asterisk (*) can have different meanings in a Java program:

- an asterisk can be used to indicate multiplication.

```
int product = 3 * 4;
```

- an asterisk can be used to mark the beginning or end of a multi-line comment.

```
/*
 * This is a comment.
 * It is several lines long.
 * Only the first and last asterisks are necessary.
 */
```

- an asterisk can be used to mark the beginning or end of a Javadoc comment.

```
/**
 * This is a method that doubles a value.
 *
 * @param n is the value to be doubled.
 * @return 2 x n
 */
public int twice( int n ) {
    return 2 * n;
} // twice( int )
```

The ampersand (&) and pipe (|) characters can also have different meanings in a Java program.

- a single & means bit-wise *and*

```
int a = 6; // in binary, 6 is 0110
int b = 3; // in binary, 3 is 0011
int c = a & b; // c is 2, because 0110 & 0011 = 0010
```

- a double & means logical *and*

```
if( 18 <= age && age <= 22 ) {
    traditionalCollegeAge = true;
} // if
```

- a single | means bit-wise *or*

```
int a = 6; // in binary, 6 is 0110
int b = 3; // in binary, 3 is 0011
int c = a | b; // c is 2, because 0110 | 0011 = 0010
```

- a double | means logical (inclusive) *or*

```
if( age < 18 || age > 22 ) {
    traditionalCollegeAge = false;
} // if
```

Lesson 2

Why Java?

- describe important features of Java programming language
  - designed from the start to support the object-oriented discipline for creating software
  - a very widely used language
  - strong backing by major companies
  - now more than 20 years old
  - has evolved and continues to evolve (adding features found in more recently designed languages)
  - can be used to write many kinds of software (desktop applications, applications for the Web, mobile applications, embedded applications, enterprise applications)
- history of programming languages
  - first high level language was FORTRAN in 1954-1956
  - FORTRAN = "FORmula TRANslation", created at IBM
  - procedural languages (1960s) — assignment statements, for loops, while loops, if statements, variables, arithmetic and logic
    - short variable names
    - every statement begins at left edge of page
    - other procedural languages: BASIC, COBOL, PL/I
  - structured programming languages — C, Pascal (1970s)
    - long names for variables and functions
    - use indentation and blank lines to make program easier to read
    - less use of GOTO statements — just a few ways of repeating actions and choosing between alternative actions

    - write for 2 audiences: computer and other people
- abstract data type languages (1980s) — Ada
    - bundle description of data and definitions of functions for working with the data
    - (Ada Lovelace worked with Charles Babbage in 1800s in England —- Babbage tried to build a computer way back then! She wrote programs for the computer!)
- object oriented languages (1990s) — C++, Java, C#
    - define relationships among abstract data types (classes)
    - "inheritance"
    - write a specialized kind of abstract data type without having to rewrite parts

Lesson 3

Parts of a Java Program

Explain significance of each part of a Java program that prints a message.

Here's the code that I want you to annotate:

3.1 Source code: Simple.java

```
package apworkshop;

public class Simple {

    public static void main( String [] args ) {
        int n = Integer.parseInt( args[0] );

        n = n + 1;

        System.out.println( Integer.toString(n) );
        System.out.println( n );

        System.out.println( "Thank you " + n + " times!" );
    } // main( String [] )

} // Simple
```

Here's a solution to the problem:

3.2 Source code: SimpleAnnotated.java

```
// This is a comment.
```

```java
// A package is a collection of related classes.
// Classes are related, for example, if they are
// part of the same project.

// It is possible to write Java programs without
// creating a package, but it is good form and good
// practice to place classes in packages, even when
// writing very simple programs.

// The package statement must be the first statement
// in a program.
package apworkshop;

// A "class" is a collection of related variables
// and methods.
// Every Java program contains at least one class.
// Classes are the most fundamental unit of organization
// Java programs.
// By convention, the names of classes are capitalized.
public class SimpleAnnotated {

    // The computer has to know where to begin
    // executing a program.
    // In a Java application, execution will always begin in a
    // function named "main."

    // The main() function can have parameters that can be used
    // be used to read information from the command line.
    // We will rarely use this feature of the language.
    // This example shows how to use the feature just so you
    // can see what the "String [] args" is all about.

    // In every one of your programs, you will have a line
    // that looks exactly like this one. The only part that
    // you may change is "args." You can give that parameter
    // any name that you like: "aardvark" works just as well
    // as "args."
    public static void main( String [] args ) {

        // args is an array of Strings.
        // An array is a collection of elements, each
        // of which can be referred to by its position
        // in the collection: there may be an element #0,
        // an element #1, and element #2, and so on.
```

```
        // args is an array of Strings.

        // A String is a sequence of characters (letters,
        // digits, punctuation, and so on).

        // The String "10" is not the same as the number 10.
        // "10" is a sequence of two digits: "1" and "0".
        // 10 is a numerical value that can be represented
        // in many ways: 10, "ten," X, or (in German) "zehn."

        // Integer is the name of a class that contains a method
        // that converts a String to its numerical value.
        int n = Integer.parseInt( args[0] );

        // Increase the value of n.
        n = n + 1;

        // n is an integer (a whole number), not a String.
        // We can convert an integer to a String by using
        // the toString() method of the Integer class:
        System.out.println( Integer.toString(n) );

        // The designers of the Java programming language
        // have given us a shortcut for converting a number
        // to a String.
        // The println() method expects a String.
        // I have given it an integer.
        // Java automatically converts the integer
        // to a String in this case.
        System.out.println( n );

        // "System" is the name of a class.

        // "out" is the name of a variable within the System class.
        // "out" is a reference to an instance of another class.
        // That other class has a method named "println."

        // "println" expects just one argument.
        // That argument must be a String.
        // This code builds the argument by concatenating
        // (pasting together) three Strings: "Thank you ",
        // the String representation of n, and " times!".
        System.out.println( "Thank you " + n + " times!" );
    } // main( String[] )

} // SimpleAnnotated
```

Lesson 4

Variables

- a variable in mathematics is the unknown quantity
- a variable in physics is the changing quantity
- a variable in computer science is **a named location in the computer's memory**
- a variable is completely defined by 6 properties
  - a name
  - an address
  - a value
  - a type
    - how much memory is needed
    - how will the value be encoded
    - what kinds of operations are permitted
  - a scope—**where** is the variable usable
  - a lifetime—**when** is the variable usable

4.1 The rules are different.

Look up the populations of China and India on the Internet.

Write a program that computes the sums of the populations of the two countries using integer variables only.

4.2 Source code: Asia.java

```
package apworkshop;

public class Asia {
    public static void main( String [] args ) {
        int china = 0; // replace 0 with the population of China
        int india = 0; // replace 0 with the population of India

        int asia = china + india;

        System.out.println( "The population of Asia is at least " +
            asia );
    } // main( String [] )
} // Asia
```

Run the program. What happened? Why?

4.3 Primitive types and reference types

4.4 Source code: ParameterPassing.java

```
package apworkshop;

public class ParameterPassing {

    public static class WholeNumber {
        private int value;

        public WholeNumber( int value ) {
            this.value = value;
        } // WholeNumber( int )

        public int getValue() {
            return this.value;
        } // getValue()

        public void setValue( int value ) {
            this.value = value;
        } // setValue( int )
    } // WholeNumber

    public static void swapA( int a, int b ) {
        int temp = a;
```

```
        a = b;
        b = temp;
    } // swapA( int, int )

    public static void swapB( WholeNumber a, WholeNumber b ) {
        int temp = a.getValue();
        a.setValue( b.getValue() );
        b.setValue( temp );
    } // swapB( WholeNumber, WholeNumber )

    public static void swapC( int[] data ) {
        int temp = data[0];
        data[0] = data[1];
        data[1] = temp;
    } // swapC( int[] )

    public static void main( String[] args ) {

        // primitive type
        int x = 17;
        int y = 19;

        System.out.println( "\nBefore swap (primitive type)..." );
        System.out.println( "x = " + x );
        System.out.println( "y = " + y );

        swapA( x, y );

        System.out.println( "After swap..." );
        System.out.println( "x = " + x );
        System.out.println( "y = " + y );

        // reference type
        WholeNumber u = new WholeNumber( 29 );
        WholeNumber v = new WholeNumber( 31 );

        System.out.println( "\nBefore swap (reference type)..." );
        System.out.println( "u = " + u.getValue() );
        System.out.println( "v = " + v.getValue() );

        swapB( u, v );

        System.out.println( "After swap..." );
        System.out.println( "u = " + u.getValue() );
        System.out.println( "v = " + v.getValue() );
```

```
        // array type
        int [] values = { 11, 13 };

        System.out.println( "\nBefore swap (array type)..." );
        System.out.println( "values[0] = " + values[0] );
        System.out.println( "values[1] = " + values[1] );

        swapC( values );

        System.out.println( "After swap..." );
        System.out.println( "values[0] = " + values[0] );
        System.out.println( "values[1] = " + values[1] );

    } // main( String [] )

} // ParameterPassing
```

Lesson 5

Methods

Write a program that computes triangular numbers.

Here is the solution:

5.1 Source code: TriangularNumbers.java

```
package apworkshop;

public class TriangularNumbers {

    // The only references to this method will be
    // from within this class.
    // Making the method "private" prevents invocations
    // of the method from within other classes.
    private static int triangularNumber( int n ) {
        // Here is one way of computing the n-th
        // triangular number, hidden in a comment.
        /*
        int sum = 0;

        for( int i = 0; i <= n; i++ ) {
            sum = sum + i;
        } // for

        return sum;
        */

        // Here is another way of computing the n-th
```

```
        // triangular number.
        return n * (n + 1) / 2;
    } // triangularNumber( int )

    // Static is the opposite of dynamic.
    // Dynamic means "created on the fly."
    // Static means "available at the start of the program's execution."
    // If main() were dynamic, we would have to wait for the computer
    // to do something before the main() method became available to us.
    // Since main() is supposed to be the first thing that the program
    // does, it cannot be dynamic--it must be static.
    public static void main( String [] args ) {

        // Repeat an action (print the i-th triangular number) 10 times.
        // Create a variable i.
        // Start with i equal to 0.
        // Increase i by one after each action.
        // Stop when i is no longer less than 10.
        for( int i = 0; i < 10; i++ ) {
            System.out.println( "The " + i + "-th triangular number = " +
                triangularNumber( i ) );
        } // for
    } // main( String [] )

} // TriangularNumbers
```

This example contains two methods. Execution of the program begins in the main() method. The triangularNumber() method produces the value of the nth triangular number. The main() method calls triangularNumber() whenever it needs the value of a triangular number.

Here is a prescription for writing any method.

- describe function's purpose in your own language
- give function a sensible, descriptive name
- identify number and type of parameters that function needs
- identify type of value that function will return to its caller
- write the sequence of logical and arithmetic statements that will produce the result

Discuss good ways of working. Here are some ideas to get you started.

How might might you apply these ideas? How might you elaborate them? (That is, can you make the rules more specific? Would you explain them differently to a teammate? Can you give reasons to justify their use?)

- take small steps
- test frequently
- adopt conventions
- work with a partner

Lesson 6

Practice with methods: computing means

Write and test functions that compute the mean of two numbers in three different ways.

Arithmetic mean $\frac{(a+b)}{2}$

Geometric mean $\sqrt{a \cdot b}$

Harmonic mean $\frac{2}{\frac{1}{a}+\frac{1}{b}}$

6.1 Source code: Means.java

```
package apworkshop;

// Compile this program by typing:
//      javac apworkshop/Means.java
// Run this program by typing the name
// of the executable file and 2 numbers:
//      java apworkshop.Means 6 10

public class Means {

    // Compute the arithmetic mean of two numbers
    public static double arithmeticMean( double a, double b ) {
        return (a + b) / 2;
    } // arithmeticMean( double, double )
```

```
    // Compute the geometric mean of two numbers
    public static double geometricMean( double a, double b ) {
        return Math.sqrt( a * b );
    } // geometricMean( double, double )

    // Compute the harmonic mean of two numbers
    public static double harmonicMean(double a,double b){
        return 2/(1/a+1/b);
    } // harmonicMean( double, double )

    // Test the functions that compute means
    public static void main( String [] args ) {
      // Find the two numbers on the command line
      // Convert the strings to floating point numbers
      double x = Double.parseDouble( args[0] );
      double y = Double.parseDouble( args[1] );

      // Print the values of the 2 numbers whose
      // mean will be computed
      System.out.printf( "x = %8.4f\n", x );
      System.out.printf( "y = %8.4f\n", y );

      // Compute and print the arithmetic mean
      double am = arithmeticMean( x, y );
      System.out.printf( "arithmetic mean = %8.4f\n", am );

      // Compute and print the geometric mean
      double gm = geometricMean( x, y );
      System.out.printf( "geometric mean = %8.4f\n", gm );

      // Compute and print the harmonic mean
      double hm = harmonicMean( x, y );
      System.out.printf( "harmonic mean = %8.4f\n", hm );

    } // main( String [] )

} // Means
```

Lesson 7

Practice with methods: computing distances

Write a program that computes the distance between two points in the plane in two different ways.

Euclidean Distance the distance between (x_0, y_0) and (x_1, y_1) is $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$

Manhattan Distance the distance between (a Street, b Avenue) and (c Street, d Avenue) is $|a - c| + |b - d|$

7.1 Source code: Distances.java

```
package apworkshop;

// Compile this program by typing:
//      javac apworkshop/Distances.java
// Run this program by typing:
//      java apworkshop.Distances

public class Distances {

    // Compute the Euclidean distance between 2 points
    public static double euclideanDistance(double x0, double y0,
            double x1, double y1) {
        double xDiff = x0 - x1;
        double yDiff = y0 - y1;
        return Math.sqrt( xDiff * xDiff + yDiff * yDiff );
    } // euclideanDistance( double, double, double, double )
```

```
    // Compute the Manhattan distance between 2 points
    public static double manhattanDistance(double a, double b,
            double c, double d ) {
        double acDiff = a - c;
        double bdDiff = d - b;
        return Math.abs ( acDiff) + Math.abs (bdDiff);
    } // manhattanDistance( double, double, double, double )

    public static void main( String [] args ) {
        double x0 = 1.0;
        double y0 = 2.0;

        double x1 = 3.0;
        double y1 = 4.0;

        System.out.printf("The Euclidean distance is %8.4f\n",
            euclideanDistance( x0, y0, x1, y1)) ;
        System.out.printf("The Manhattan distance is %8.4f\n",
            manhattanDistance( x0, y0, x1, y1)) ;
    } // main( String [] )

} // Distances
```

Lesson 8

Practice with methods: weighted averages

Define and test a function that computes the weighted average of 2 numbers;

If a and b are the two numbers whose weighted average we want to compute and t is the weight $(0.0 \leq t \leq 1.0)$, then...

$$f(a, b, t) = (1 - t) \cdot a + t \cdot b$$

Can you also write a Java function that computes $g(a, b, c, t) = f(f(a, b, t), f(b, c, t), t)$?

Can you also write a Java function that computes $h(a, b, c, d, t) = f(g(a, b, c, t), g(b, c, d, t), t)$?

These functions can be used to compute the coordinates of points on a Bézier curve. Bézier are used in computer-aided design, illustration, and typography.

8.1 Source code: BezierFrame.java

```
package apworkshop;

import java.awt.Container;
import javax.swing.JFrame;

public class BezierFrame extends JFrame {

    private static final int FRAME_WIDTH = 512;
    private static final int FRAME_HEIGHT = 512;
    private static final String TITLE = "Bezier Curve";
```

```
    public BezierFrame() {
        this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        this.setSize( FRAME_WIDTH, FRAME_HEIGHT );
        this.setTitle( TITLE );

        Container pane = this.getContentPane();
        pane.add( new BezierPanel() );

        this.setVisible( true );
    } // BezierFrame()

    public static void main( String [] args ) {
        BezierFrame bezierFrame = new BezierFrame();
    } // main( String [] )

} // BezierFrame
```

8.2 Source code: BezierPanel.java

```
package apworkshop;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Shape;
import java.awt.geom.AffineTransform;
import java.awt.geom.Ellipse2D;
import java.awt.geom.Line2D;
import java.awt.geom.Point2D;
import java.util.Random;
import javax.swing.JPanel;

public class BezierPanel extends JPanel {

    private static final Color BG_COLOR =
        new Color( 172, 248, 206 );

    private static final Color VERTEX_COLOR =
        new Color( 248, 112, 144 );

    private static final Color POLYGON_COLOR =
        new Color( 160, 180, 248 );
```

```
    private static final Color CURVE_COLOR =
        new Color( 224, 248, 128 );

    private static final double DOT_RADIUS = 0.02;
    private static final float CURVE_WIDTH = 4F;

    private static final int NUMBER_OF_SEGMENTS = 128;

    private final Random rng = new Random();

    public BezierPanel() {
        this.setBackground( BG_COLOR );
    } // BezierPanel()

    @Override
    public void paintComponent( Graphics g ) {
        super.paintComponent( g );
        Graphics2D g2D = (Graphics2D) g;

        int w = this.getWidth();
        int h = this.getHeight();

        // All geometry will be drawn in a square
        // whose lower left corner is at (-1,-1)
        // and whose upper right corner is at (+1,+1).
        // AffineTransforms will move and enlarge the
        // geometry so that it fits within the JPanel.
        AffineTransform translate = new AffineTransform();
        translate.setToTranslation( 1, 1 );

        AffineTransform scale = new AffineTransform();
        scale.setToScale( w/2, h/2 );

        AffineTransform transform = new AffineTransform();
        transform.concatenate( scale );
        transform.concatenate( translate );

        // Construct control points.
        Point2D [] points = new Point2D[4];
        for( int i = 0; i < points.length; i++ ) {
            double x = 2 * rng.nextDouble() - 1;
            double y = 2 * rng.nextDouble() - 1;

            points[i] = new Point2D.Double( x, y );
        } // for
```

```
// Draw line segments that connect the
// control points.
g2D.setColor( POLYGON_COLOR );
g2D.setStroke( new BasicStroke( CURVE_WIDTH ) );

for( int i = 0; i < points.length; i++ ) {
    double x0 = points[i].getX();
    double y0 = points[i].getY();

    int j = (i + 1) % points.length;

    double x1 = points[j].getX();
    double y1 = points[j].getY();

    Line2D line = new Line2D.Double( x0, y0, x1, y1 );
    Shape shape = transform.createTransformedShape( line );

    g2D.draw( shape );
} // for


g2D.setColor( CURVE_COLOR );

// Draw a sequence of line segments
// that approximate the Bezier curve.
double fraction = 1.0 / NUMBER_OF_SEGMENTS;
for( int i = 0; i < NUMBER_OF_SEGMENTS; i++ ) {
    double t0 = i * fraction;
    double t1 = (i + 1) * fraction;

    Point2D p0 = weightedAverage( points[0], points[1],
        points[2], points[3], t0 );
    Point2D p1 = weightedAverage( points[0], points[1],
        points[2], points[3], t1 );

    double x0 = p0.getX();
    double y0 = p0.getY();
    double x1 = p1.getX();
    double y1 = p1.getY();

    Line2D line = new Line2D.Double( x0, y0, x1, y1 );
    Shape shape = transform.createTransformedShape( line );

    g2D.draw( shape );
} // for
```

```
        // Draw the control points.
        g2D.setColor( VERTEX_COLOR );

        for( Point2D p : points  ) {
            double xMin = p.getX() - DOT_RADIUS;
            double yMin = p.getY() - DOT_RADIUS;
            double diameter = 2 * DOT_RADIUS;

            Ellipse2D dot = new Ellipse2D.Double(
                xMin, yMin, diameter, diameter );

            Shape shape = transform.createTransformedShape( dot );
            g2D.fill( shape );
        } // for

    } // paintComponent( Graphics )

    /**
      * Compute the weighted average of two numbers.
      *
      * @param a is one of the two numbers.
      * @param b is the other number.
      * @param t is the weight (a value in the interval [0,1]).
      * @return the weighted average of a and b, with weight t.
      */
    private double weightedAverage( double a, double b, double t ) {
        return (1 - t) * a + t * b;
    } // weightedAverage( double, double, double )

    /**
     * Compute the weighted average of two points.
     *
     * @param p0 is one of the two points.
     * @param p1 is the other point.
     * @param t is the weight (a value in the interval [0,1]).
     * @return a point whose x coordinate is the weighted
     *   average of the x coordinates of p0 and p1, and whose
     *   y coordinate is the weighted average of the y coordinates
     *   of p0 and p1, with weight t.
     */
    private Point2D weightedAverage( Point2D p0, Point2D p1, double t ) {
        double x0 = p0.getX();
        double y0 = p0.getY();

        double x1 = p1.getX();
```

```
        double y1 = p1.getY();

        double x = weightedAverage( x0, x1, t );
        double y = weightedAverage( y0, y1, t );

        return new Point2D.Double( x, y );
    } // weightedAverage( Point2D, Point2D, double )

    /**
     * Compute the weighted average of the weighted averages
     * of two pairs of points.
     *
     * @param p0 is the first point in the first pair of points.
     * @param p1 is the second point in the first pair
     *   of points and the first point in the second
     *   pair of points.
     * @param p2 is the second point in the second pair
     *   of points.
     * @param t is the weight (a value in the interval [0,1]).
     * @return the weighted average of weighted averages.
     */
    private Point2D weightedAverage( Point2D p0, Point2D p1,
          Point2D p2, double t ) {

        Point2D p01 = weightedAverage( p0, p1, t );
        Point2D p12 = weightedAverage( p1, p2, t );

        return weightedAverage( p01, p12, t );
    } // weightedAverage( Point2D, Point2D, Point2D, double )

    /**
     * Compute the weighted average of the weighted
     * averages of weighted averages of points.
     *
     * @param p0 is the first point in the first
     *   triplet of points.
     * @param p1 is the second point in the first
     *   triplet of points and the first point in
     *   the second triplet.
     * @param p2 is the third point in the first
     *   triplet of points and the second point in
     *   the second triplet.
     * @param p3 is the third point in the second
     *   triplet of points.
     * @param t is the weight (a value in the interval [0,1]).
     * @return a weighted average of weighted averages of
```

```
     *    weighted averages.
     */
    private Point2D weightedAverage( Point2D p0, Point2D p1,
          Point2D p2, Point2D p3, double t ) {

        Point2D p012 = weightedAverage( p0, p1, p2, t );
        Point2D p123 = weightedAverage( p1, p2, p3, t );

        return weightedAverage( p012, p123, t );
    } // weightedAverage( Point2D, Point2D, Point2D, Point2D, double )
} // BezierPanel
```

Lesson 9

My favorite definition of computer science

We can define computer science with four questions.

1. Which problems can we solve with a computer program?

   This is the problem of **computability**.

   It is possible to prove the impossibility of ever writing a computer program that will answer some questions.

   A principal example of this is the Halting Problem, given to us by Alan Turing.

2. Given a problem that we can solve, how can we write the program that solves it?

   A study of **software engineering** will help us here.

3. Given two or more programs that solve a problem, how can we choose the best program?

   An examination of **complexity** will help us answer this question.

4. Given a program and a claim that it solves a problem, how can we be sure that it will produce the right answer every time?

   This is the examination of **correctness**.

9.1 A first look at complexity: bank accounts

Let's suppose that you have several accounts at a bank. For example, you might have a savings account, a checking account, a credit card account, a loan for an

automobile, and a mortgage.

Each of these accounts has a unique integer identifier.

You have records of your withdrawals and deposits for each of these accounts.

You would like a program that will read these records and update the balance for each of your accounts.

Suppose that the accounts are numbered. Each input record will identify an account by number and the amount of the deposit (a positive value) or withdrawal (a negative value).

Here's a start to the solution of this problem.

9.1.1 Source code: NumberedAccounts.java

```
package apworkshop;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class NumberedAccounts {

    private static final int NUMBER_OF_ACCOUNTS = 5;

    public static void main( String [] args ) {

        try {
            double accounts[] = new double[ NUMBER_OF_ACCOUNTS ];

            File file = new File( "accounts-numbers.txt" );
            Scanner scanner = new Scanner( file );

            while( scanner.hasNext() ) {
              int accountNumber = scanner.nextInt();
              double amount = scanner.nextDouble();
              accounts[ accountNumber ] += amount;
            } // while

            scanner.close();

            for( int i = 0; i < NUMBER_OF_ACCOUNTS; i++ ) {
                System.out.printf( "Account #%2d has a balance of %6.2f\n",
                    i, accounts[i] );
            } // for
```

```
        } // try
        catch( FileNotFoundException e ) {
            e.printStackTrace();
        } // catch( FileNotFoundException )

    } // main( String [] )

} // NumberedAccounts
```

9.1.2 Sample input

```
0 275.31
1 -250
0 125.43
3 -23.59
4 -60.42
3 -18.07
```

9.1.3 What to do next?

- How can we read data from a file?
- How can we write our report to a file?
- Could we identify accounts by name instead of by number?
- How can we measure how much work the program does?

Now suppose that the accounts are named rather than numbered.

9.1.4 Source code: NamedAccounts.java

```
package apworkshop;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.HashMap;
import java.util.Scanner;

public class NamedAccounts {

    private static HashMap<String,Double> accounts = new HashMap<>();

    public static void main( String [] args ) {
```

```
        try {

            File file = new File( "accounts-names.txt" );
            Scanner scanner = new Scanner( file );

            while( scanner.hasNext() ) {
              String accountName = scanner.next();
              double amount = scanner.nextDouble();

              if( accounts.get( accountName ) == null ) {
                  accounts.put( accountName, amount );
              } // if
              else {
                  double oldValue = accounts.get( accountName );
                  accounts.put( accountName, oldValue + amount );
              } // else

            } // while

            scanner.close();


            for( String key : accounts.keySet() ) {
                System.out.printf( "Account \"%s\" has a balance of %6.2f\n"
                    key, accounts.get(key) );
            } // for

        } // try
        catch( FileNotFoundException e ) {
            e.printStackTrace();
        } // catch( FileNotFoundException )

    } // main( String [] )

} // NamedAccounts
```

9.1.5 Sample input

```
earn 275.31
rent -250
earn 125.43
food -23.59
book -60.42
food -18.07
```

Lesson 10

Reading and writing files.

10.1 Source code: FileIO.java

```
package apworkshop;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class FileIO {

    public static void main( String [] args ) {

        try {
            File inputFile = new File( "computer-scientists-in.txt" );
            Scanner scanner = new Scanner( inputFile );

            File outputFile = new File( "computer-scientists-out.txt" );
            FileWriter fileWriter = new FileWriter( outputFile );

            while( scanner.hasNext() ) {
                String firstName = scanner.next();
                String lastName = scanner.next();

                System.out.println( lastName + ", " + firstName );
                fileWriter.append( lastName + ", " + firstName + "\n" );
            } // while

            scanner.close();
```

```
            fileWriter.close();
        } // try
        catch( IOException e ) {
            e.printStackTrace();
        } // catch( FileNotFoundException )

    } // main( String [] )

} // FileIO
```

Lesson 11

Traces of sorts

Both the selection sort and the insertion sort are "simple sorts." They are also called "quadratic time sorts." This is because the two algorithms require a time that is proportional to the square of the length of the list that they sort in the average and worst cases. However, in the average case the insertion sort is faster than the selection sort (the proportionality constant is smaller) and in the best case the insertion sort only requires a time that is proportional to the length of the list (not its square).

There is an algorithm for searching inside both of these algorithms for sorting.

There are other algorithms that can sort more quickly than can the selection sort or insertion sort algorithms.

11.1 Trace of selection sort

At Step k the first k elements of the list are in order.

At Step k the smallest k elements of the list are at the front of the list.

At Step k in a sort of N items the algorithm must examine $N - k$ elements.

At each step, the algorithm searches from left to right through the unsorted part of the list. It seeks the smallest element in that part of the list. In order to be sure that it has found the smallest element, it must examine every element in that part of the list.

The selection sort does the same amount of work when given a nearly sorted list as it does when given a thoroughly shuffled list.

Here is an example. The the numbers that appear in a **boldface** font are the elements in the already sorted part of the list. As you can see, at each step the number of elements in the already sorted part increases by one.

Step												
0	32	43	89	21	96	40	76	11	35	43	71	97
1	**11**	43	89	21	96	40	76	32	35	43	71	97
2	**11**	**21**	89	43	96	40	76	32	35	43	71	97
3	**11**	**21**	**32**	43	96	40	76	89	35	43	71	97
4	**11**	**21**	**32**	**35**	96	40	76	89	43	43	71	97
5	**11**	**21**	**32**	**35**	**40**	96	76	89	43	43	71	97
6	**11**	**21**	**32**	**35**	**40**	**43**	76	89	96	43	71	97
7	**11**	**21**	**32**	**35**	**40**	**43**	**43**	89	96	76	71	97
8	**11**	**21**	**32**	**35**	**40**	**43**	**43**	**71**	96	76	89	97
9	**11**	**21**	**32**	**35**	**40**	**43**	**43**	**71**	**76**	96	89	97
10	**11**	**21**	**32**	**35**	**40**	**43**	**43**	**71**	**76**	**89**	96	97
11	**11**	**21**	**32**	**35**	**40**	**43**	**43**	**71**	**76**	**89**	**96**	97
12	**11**	**21**	**32**	**35**	**40**	**43**	**43**	**71**	**76**	**89**	**96**	**97**

11.2 Trace of insertion sort

At Step k the first k elements of the list are in order.

At Step k the algorithm must search from right to left through the already sorted part of the list to find the place at which to insert the next element from the unsorted part of the list.

At Step k the algorithm must examine between 0 and k elements.

At each step, the algorithm searches from right to left through the already sorted part of the list. It stops as soon as it finds an element that is less than or equal to the first element in the unsorted part of the list. This could be the very first element it checks, so it does not always have to check every element in the unsorted part of the list.

The insertion sort does less work when given a list that is nearly sorted than it does when given a list that is thoroughly shuffled.

In practice, many of the lists that we are asked to sort are already partly sorted.

Here is an example. The the numbers that appear in a **boldface** font are the elements in the already sorted part of the list. As you can see, at each step the number of elements in the already sorted part increases by one.

Step												
0	**57**	26	41	01	74	22	48	99	63	63	67	40
1	**26**	**57**	41	01	74	22	48	99	63	63	67	40
2	**26**	**41**	**57**	01	74	22	48	99	63	63	67	40
3	**01**	**26**	**41**	**57**	74	22	48	99	63	63	67	40
4	**01**	**26**	**41**	**57**	**74**	22	48	99	63	63	67	40
5	**01**	**22**	**26**	**41**	**57**	**74**	48	99	63	63	67	40
6	**01**	**22**	**26**	**41**	**48**	**57**	**74**	99	63	63	67	40
7	**01**	**22**	**26**	**41**	**48**	**57**	**74**	**99**	63	63	67	40
8	**01**	**22**	**26**	**41**	**48**	**57**	**63**	**74**	**99**	63	67	40
9	**01**	**22**	**26**	**41**	**48**	**57**	**63**	**63**	**74**	**99**	67	40
10	**01**	**22**	**26**	**41**	**48**	**57**	**63**	**63**	**67**	**74**	**99**	40
11	**01**	**22**	**26**	**40**	**41**	**48**	**57**	**63**	**63**	**67**	**74**	**99**

11.3 Trace of merge sort

Here's a list of random numbers. If we are lucky, the numbers are already in order.

46	39	36	07	90	32	78	25	88	38	62	09	89	83	79	75

If we are not so lucky, let's divide the list in half. If we are lucky, the numbers in the first half are already in order and the numbers in the second half are

already in order.

| 46 39 36 07 90 32 78 25 | 88 38 62 09 89 83 79 75 |

If we are not lucky, let's divide the halves in half. Are those smaller pieces already sorted?

| 46 39 36 07 | 90 32 78 25 | 88 38 62 09 | 89 83 79 75 |

Divide again.

| 46 39 | 36 07 | 90 32 | 78 25 | 88 38 | 62 09 | 89 83 | 79 75 |

A sublist that has only one element cannot be divided further.

| 46 | 39 | 36 | 07 | 90 | 32 | 78 | 25 | 88 | 38 | 62 | 09 | 89 | 83 | 79 | 75 |

| 46 | 39 | 36 | 07 | 90 | 32 | 78 | 25 | 88 | 38 | 62 | 09 | 89 | 83 | 79 | 75 |

Sort the pairs, swapping adjacent numbers where necessary.

| 39 46 | 07 36 | 32 90 | 25 78 | 38 88 | 09 62 | 83 89 | 75 79 |

Merge adjacent lists of two numbers to produce lists of four numbers.

| 07 36 39 46 | 25 32 78 90 | 09 38 62 88 | 75 70 83 89 |

Merge adjacent lists of four numbers to produce lists of eight numbers.

| 07 25 32 36 39 46 78 90 | 09 38 62 70 75 83 88 89 |

Finally, merge adjacent lists of eight numbers to produce a list of sixteen numbers.

We are done.

| 07 09 25 32 36 38 39 46 62 75 78 79 83 88 89 90 |

Lesson 12

Selection sort

We will learn a simple algorithm for sorting through a sequence of exercises.

Here is a program that shows how to generate random numbers.

12.1 Source code: RandomNumbers.java

```
package apworkshop;

import java.util.Random;

/**
 * A throw-away program to experiment with the
 * methods of the Random class.
 *
 * @author Susan Calvin
 * @version 14 July 2018
 */
public class RandomNumbers {

    /**
     * MINIMUM is the smallest random integer that
     * this program can generate.
     */
    private static final int MINIMUM = 10;

    /**
     * MAXIMUM is the largest random integer plus one that
     * this program can generate.
```

```
     */
    private static final int MAXIMUM = 20;

    /**
     * COUNT is the number of random integers that
     * this program will generate.
     */
    private static final int COUNT = 12;

    /**
     * The Random class has methods for generating
     * random integers and random floating point values.
     */
    private static final Random RNG = new Random();

    public static void main( String [] args ) {

        for( int i = 0; i < COUNT; i++ ) {
            // generate a random integer value between minimum and maximum -
            int value = MINIMUM + RNG.nextInt( MAXIMUM - MINIMUM );

            System.out.printf( "A random integer = %4d\n", value );
        } // for

    } // main( String[] )
} // RandomNumbers
```

- Write a function that produces an array of random integers whose values lie within specified bounds.
- Write a function that finds the smallest integer in an array of integers.
- Write a function that finds the position of the smallest integer in an array of integers.
- Write a function that finds the position of the smallest integer in that part of an array of integers that begins at a specified position.
- Write a function that exchanges the values at two specified positions within an array of integers.
- Write a function that sorts the integers in an array using the selection sort algorithm.
- Add code to the selection sort program that will count the number of times that the program compares one integer with another.

12.2 Source code for exercise: Sorts.java

```
package apworkshop;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

/**
 * An exercise with sorting algorithms.
 *
 * Complete the stub methods.
 *
 * Here is an example of output from the
 * the completed program:
 *
 *
 * Partly sorted list.
 *  14  17  34  48  63  76  82  99  16  33  72  59  28  20
13  49
 *
 * Sorted list.
 *   1   6  14  14  20  23  42  43  50  58  64  69  74  79
87  98
 *
 * After swapping elements 3 and 5, the list is:
 *   1   6  14  23  20  14  42  43  50  58  64  69  74  79
87  98
 *
 * Unsorted list.
 *  93  18  35  13  33  28  73  66  91   8  44  86  24  70
20  39
 *
 * Value of smallest integer in list = 8
 *
 * Position of smallest integer in list = 9
 *
 * Position of smallest integers in that
 *  part of the list that begins at index 8 = 9
 *
 * After sorting with the selection sort:
 *   8  13  18  20  24  28  33  35  39  44  66  70  73  86
91  93
 *
```

```
 *   26   34   36   59   61   65   83   90   50   23    5   56   17   50
96   16
 * The element at index 8 belongs at position 3
 *
 * Here is the list with a hole created at the right place.
 *   26   34   36 null   59   61   65   83   90   23    5   56   17   50
96   16
 *
 * Here an unsorted list.
 *   11   27   22   64   73   76   63   24   10   54   24   14   45   34
11   54
 * And here is the same list after sorting with insertion sort.
 *   10   11   11   14   22   24   24   27   34   45   54   54   63   64
73   76
 *
 * Here is a list whose two parts are sorted.
 *   20   34   43   48   50   51   55   86    1   36   61   76   81   86
88   95
 *
 * Here is the list after the merge of the two parts.
 *    1   20   34   36   43   48   50   51   55   61   76   81   86   86
88   95
 *
 * @author Your Name
 * @version 02 April 2018
 */
public class Sorts {

    /**
     * The maximum number of integers that
     * should put on a line of output.
     */
    public static final int INTEGERS_ON_A_LINE = 16;

    /**
     * The largest non-negative value that a random
     * number (to be added to a list) may assume.
     */
    public static final int MAX_VALUE = 99;

    /**
     * A random number generator.
     */
    public static final Random rng = new Random();

    /**
```

```
     * Create a list of non-negative random numbers whose
     * first part is sorted.
     *
     * @param sortedSize is the number of elements (a non-negative integer)
     * in the sorted part of the list
     * @param totalSize is the total number of elements (a non-negative integer)
     * in the list
     * @param maxValue is the largest value that any
     * of the non-negative random numbers that will be
     * generated and added to the list
     * @return is the list of random integers
     */
    public static List<Integer> makePartlySortedList( int sortedSize,
            int totalSize, int maxValue ) {

        List<Integer> result = new ArrayList<>();

        return result;
    } // makePartlySortedList( int, int, int )

    /**
     * Create a list of non-negative integers.
     *
     * @param size is the number of integers in the list.
     * @param maxValue is the largest value that any of
     * the non-negative random numbers that are generated
     * and added to the list can assume
     * @return is the list of random integers
     */
    public static List<Integer> makeSortedList( int size, int maxValue ) {

        List<Integer> result = new ArrayList<>();

        return result;
    } // makeSortedList( int, int )

    /**
     * Create a list the contains random non-negative
     * integers and is divided into two parts, both
     * of which are sorted
     *
     * @param prefixSize is the number of integers in
     * the first part of the list
     * @param suffixSize is the number of integers in
     * the second part of the list
     * @return the list of random integers with its
```

```
 * first and second parts sorted
 */
public static List<Integer> makeTwoPartList( int prefixSize,
        int suffixSize, int maxValue ) {
    List<Integer> result = new ArrayList<>();

    return result;
} // makeTwoPartList( int, int, int )

/**
 * Create an unsorted list of random, non-negative integers.
 *
 * @param size is the number of integers in the list
 * @param maxValue is the largest non-negative value that
 * any of the random integers may assume
 * @return the list of integers
 */
public static List<Integer> makeUnsortedList( int size, int maxValue ) {
    List<Integer> result = new ArrayList<>();

    return result;
} // makeUnsortedList( int )

/**
 * Find the smallest integer in a list of integers
 *
 * @param data is the list in which to sort
 * @return the smallest integer in the list
 */
public static int minimum( List<Integer> data ) {
    return 0;
} // minimum( List<Integer> )

/**
 * Find the index of the smallest integer in a list
 * of integers
 *
 * @param data is the list in which to sort
 * @return the index of the smallest integer in the list
 */
public static int positionOfMinimum( List<Integer> data ) {
    return 0;
} // positionOfMinimum( List<Integer> )

/**
 * Find the index of the smallest integer in that
```

```
     * part of a list of integers that begins at a specified
     * position
     *
     * @param startingIndex is the position of that part of the
     * list in which to begin the search
     * @param data is the list in which to search
     * @return the index of the smallest integer in that part
     * the list that begins the specified position
     */
    public static int positionOfMinimum( int startingIndex, List<Integer> data ) {
        return 0;
    } // positionOfMinimum( List<Integer> )

    /**
     * Exchange the values of the elements at
     * positions i and j in a list.
     *
     * @param i is an index in the list and in the
     * interval [0, data.size() - 1]
     * @param j is an index in the list and in the
     * interval [0, data.size() - 1]
     * @param data is the list that contains the
     * two elements
     */
    public static void swap( int i, int j, List<Integer> data ) {

    } // swap( int, int, List<Integer> )

    /**
     * Sort a list of integers using the selection sort algorithm.
     *
     * @param data is the list to be sorted
     */
    public static void selectionSort( List<Integer> data ) {

    } // selectionSort( List<Integer> )

    /**
     * Find the position in the sorted part of a list
     * in which to insert the next element.
     *
     * This will be the last element (in a search from right
     * to left) that is greater than or equal to a given element
     *
     * @param index is the position at which to begin the left to
     * right search and is also the position of the element
```

```
 * to which all other elements in the search will be compared
 * @param data is the list in which to search
 * @return the index of the insertion point
 */
public static int positionOfGE( int index, List<Integer> data ) {

    return 0;
} // positionOfGE( int, List<Integer> )

/**
 * Create a hole in a list into which to insert
 * an element in the next step of an insertion sort.
 *
 * @param i is the index of the element to be inserted
 * (0 <= j <= i <= data.size - 1)
 * @param j is the position at which the element is to
 * be inserted (0 <= j <= i <= data.size - 1)
 */
public static void createHole( int i, int j, List<Integer> data ) {

} // createHole( int, int,  List<Integer> )

/**
 * Sort a list using the insertion sort algorithm.
 *
 * @param data is the list to be sorted
 */
public static void insertionSort( List<Integer> data ) {

} // insertionSort( List<Integer> )

/**
 * Merge the two parts of a list (when those
 * two parts are already sorted)
 *
 * The two parts of the list will contain
 * consecutive elements elements of the list.
 *
 * The two parts need not make up the whole
 * list--there may be other elements before and/or
 * after these two parts
 *
 * @param prefixStart is the index of the first
 * element in the first part of the list
 * @param suffixStart is the index of the first
 * element in the second part of the list
```

```
     * @param suffixEnd is the end of the list
     * element in the second part of the list
     * @param data is the list
     * @return the original list with the specified
     * two parts now sorted
     */
    public static void merge( int prefixStart,
            int suffixStart, int suffixEnd, List<Integer> data ) {

    } // merge( List<Integer>, List<Integer> )

    /**
     * Sort a list using the merge sort algorithm.
     *
     * @param data is the list to be sorted.
     */
    public static void mergeSort( List<Integer> data ) {

    } // mergeSort( List<Integer> )

    public static void printList( List<Integer> list ) {
        for( int i = 0; i < list.size(); i++ ) {
            System.out.printf( "%3d ", list.get(i) );
            if( i != 0 && i % INTEGERS_ON_A_LINE == 0 ) {
                System.out.println();
            } // if
        } // for
        System.out.println();
    } // printList( List<Integer> )

    public static void main( String [] args ) {

        List<Integer> a = makePartlySortedList( 8, INTEGERS_ON_A_LINE, MAX_VALUE );
        List<Integer> b = makeSortedList( INTEGERS_ON_A_LINE, MAX_VALUE );
        List<Integer> c = makeUnsortedList( INTEGERS_ON_A_LINE, MAX_VALUE );

        System.out.println( "Partly sorted list." );
        printList( a );
        System.out.println();

        System.out.println( "Sorted list." );
        printList( b );
        System.out.println();

        System.out.println( "After swapping elements 3 and 5, the list is:" );
        swap( 3, 5, b );
```

```
printList( b );
System.out.println();

System.out.println( "Unsorted list." );
printList( c );
System.out.println();

System.out.println( "Value of smallest integer in list = " +
    minimum( c ) );
System.out.println();

System.out.println( "Position of smallest integer in list = " +
    positionOfMinimum( c ) );
System.out.println();

System.out.println( "Position of smallest integers in that \n" +
    " part of the list that begins at index 8 = " +
    positionOfMinimum( 8, c ) );
System.out.println();

System.out.println( "After sorting with the selection sort: " );
selectionSort( c );
printList( c );
System.out.println();

a = makePartlySortedList( 8, INTEGERS_ON_A_LINE, MAX_VALUE );
printList( a );
int j = positionOfGE( 8, a );
System.out.println( "The element at index 8 belongs at position " +
System.out.println();

System.out.println( "Here is the list with a hole created at the rig
createHole( 8, j, a );
printList( a );
System.out.println();

c = makeUnsortedList( INTEGERS_ON_A_LINE, MAX_VALUE );
System.out.println( "Here an unsorted list." );
printList( c );
insertionSort( c );
System.out.println( "And here is the same list after sorting with in
printList( c );
System.out.println();

List<Integer> d = makeTwoPartList( 8, 8, MAX_VALUE );
System.out.println( "Here is a list whose two parts are sorted." );
```

```
        printList( d );
        System.out.println();

        merge( 0, 8, d.size() - 1, d );
        System.out.println( "Here is the list after the merge of the two parts." );
        printList( d );

    } // main( String [] )

} // Sorts
```

12.3 Source code for solution to exercise: SortsSolution.java

```
package apworkshop;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

/**
 * An exercise with sorting algorithms.
 *
 * Complete the stub methods.
 *
 * Here is an example of output from the
 * the completed program:
 *
 *
 * Partly sorted list.
 *   14  17  34  48  63  76  82  99  16  33  72  59  28  20
13  49
 *
 * Sorted list.
 *    1   6  14  14  20  23  42  43  50  58  64  69  74  79
87  98
 *
 * After swapping elements 3 and 5, the list is:
 *    1   6  14  23  20  14  42  43  50  58  64  69  74  79
87  98
 *
 * Unsorted list.
 *   93  18  35  13  33  28  73  66  91   8  44  86  24  70
20  39
```

```
 *
 * Value of smallest integer in list = 8
 *
 * Position of smallest integer in list = 9
 *
 * Position of smallest integers in that
 *  part of the list that begins at index 8 = 9
 *
 * After sorting with the selection sort:
 *    8  13  18  20  24  28  33  35  39  44  66  70  73  86
91  93
 *
 *   26  34  36  59  61  65  83  90  50  23   5  56  17  50
96  16
 * The element at index 8 belongs at position 3
 *
 * Here is the list with a hole created at the right place.
 *   26  34  36 null  59  61  65  83  90  23   5  56  17  50
96  16
 *
 * Here an unsorted list.
 *   11  27  22  64  73  76  63  24  10  54  24  14  45  34
11  54
 * And here is the same list after sorting with insertion sort.
 *   10  11  11  14  22  24  24  27  34  45  54  54  63  64
73  76
 *
 * Here is a list whose two parts are sorted.
 *   20  34  43  48  50  51  55  86   1  36  61  76  81  86
88  95
 *
 * Here is the list after the merge of the two parts.
 *    1  20  34  36  43  48  50  51  55  61  76  81  86  86
88  95
 *
 * @author Your Name
 * @version 02 April 2018
 */
public class SortsSolution {

    /**
     * The maximum number of integers that
     * should put on a line of output.
     */
    public static final int INTEGERS_ON_A_LINE = 16;
```

```
/**
 * The largest non-negative value that a random
 * number (to be added to a list) may assume.
 */
public static final int MAX_VALUE = 99;

/**
 * A random number generator.
 */
public static final Random rng = new Random();

public static int twice( int n ) {
    return n + n;
} // twice( int )

/**
 * Create a list of non-negative random numbers whose
 * first part is sorted.
 *
 * @param sortedSize is the number of elements (a non-negative integer)
 * in the sorted part of the list
 * @param totalSize is the total number of elements (a non-negative integer)
 * in the list
 * @param maxValue is the largest value that any
 * of the non-negative random numbers that will be
 * generated and added to the list
 * @return is the list of random integers
 */
public static List<Integer> makePartlySortedList( int sortedSize,
        int totalSize, int maxValue ) {

    List<Integer> result = new ArrayList<>();

    for( int i = 0; i < sortedSize; i++ ) {
        result.add( rng.nextInt( maxValue + 1 ) );
    } // for

    Collections.sort( result );

    for( int i = sortedSize; i < totalSize; i++ ) {
        result.add( rng.nextInt( maxValue + 1 ) );
    } // for

    return result;
} // makePartlySortedList( int, int, int )
```

```
/**
 * Create a list of non-negative integers.
 *
 * @param size is the number of integers in the list.
 * @param maxValue is the largest value that any of
 * the non-negative random numbers that are generated
 * and added to the list can assume
 * @return is the list of random integers
 */
public static List<Integer> makeSortedList( int size, int maxValue ) {
    return makePartlySortedList( size, size, maxValue );
} // makeSortedList( int, int )

/**
 * Create a list the contains random non-negative
 * integers and is divided into two parts, both
 * of which are sorted
 *
 * @param prefixSize is the number of integers in
 * the first part of the list
 * @param suffixSize is the number of integers in
 * the second part of the list
 * @return the list of random integers with its
 * first and second parts sorted
 */
public static List<Integer> makeTwoPartList( int prefixSize,
        int suffixSize, int maxValue ) {
    List<Integer> prefix = makeSortedList( prefixSize, maxValue );
    List<Integer> suffix = makeSortedList( suffixSize, maxValue );

    List<Integer> result = new ArrayList<>();
    result.addAll( prefix );
    result.addAll( suffix );

    return result;
} // makeTwoPartList( int, int, int )

/**
 * Create an unsorted list of random, non-negative integers.
 *
 * @param size is the number of integers in the list
 * @param maxValue is the largest non-negative value that
 * any of the random integers may assume
 * @return the list of integers
 */
public static List<Integer> makeUnsortedList( int size, int maxValue ) {
```

```
        return makePartlySortedList( 0, size, maxValue );
    } // makeUnsortedList( int )

    /**
     * Find the smallest integer in a list of integers
     *
     * @param data is the list in which to sort
     * @return the smallest integer in the list
     */
    public static int minimum( List<Integer> data ) {
        int bestGuessSoFar = data.get(0);
        for( int i = 1; i < data.size(); i++ ) {
            if( data.get(i) < bestGuessSoFar ) {
                bestGuessSoFar = data.get(i);
            } // if
        } // for
        return bestGuessSoFar;
    } // minimum( List<Integer> )

    /**
     * Find the index of the smallest integer in a list
     * of integers
     *
     * @param data is the list in which to sort
     * @return the index of the smallest integer in the list
     */
    public static int positionOfMinimum( List<Integer> data ) {
        int bestGuessSoFar = 0;
        for( int i = 1; i < data.size(); i++ ) {
            if( data.get(i) < data.get(bestGuessSoFar) ) {
                bestGuessSoFar = i;
            } // if
        } // for
        return bestGuessSoFar;
    } // positionOfMinimum( List<Integer> )

    /**
     * Find the index of the smallest integer in that
     * part of a list of integers that begins at a specified
     * position
     *
     * @param startingIndex is the position of that part of the
     * list in which to begin the search
     * @param data is the list in which to search
     * @return the index of the smallest integer in that part
     * the list that begins the specified position
```

```
     */
    public static int positionOfMinimum( int startingIndex, List<Integer> da
        int bestGuessSoFar = startingIndex;
        for( int i = startingIndex + 1; i < data.size(); i++ ) {
            if( data.get(i) < data.get(bestGuessSoFar) ) {
                bestGuessSoFar = i;
            } // if
        } // for
        return bestGuessSoFar;
    } // positionOfMinimum( List<Integer> )

    /**
     * Exchange the values of the elements at
     * positions i and j in a list.
     *
     * @param i is an index in the list and in the
     * interval [0, data.size() - 1]
     * @param j is an index in the list and in the
     * interval [0, data.size() - 1]
     * @param data is the list that contains the
     * two elements
     */
    public static void swap( int i, int j, List<Integer> data ) {
        int temp = data.get(i);
        data.set(i, data.get(j) );
        data.set(j, temp );
    } // swap( int, int, List<Integer> )

    /**
     * Sort a list of integers using the selection sort algorithm.
     *
     * @param data is the list to be sorted
     */
    public static void selectionSort( List<Integer> data ) {
        for( int i = 0; i < data.size(); i++ ) {
            int j = positionOfMinimum( i, data );
            swap( i, j, data );
        } // for
    } // selectionSort( List<Integer> )

    /**
     * Find the position in the sorted part of a list
     * in which to insert the next element.
     *
     * This will be the last element (in a search from right
     * to left) that is greater than or equal to a given element
```

```
     *
     * @param index is the position at which to begin the left to
     * right search and is also the position of the element
     * to which all other elements in the search will be compared
     * @param data is the list in which to search
     * @return the index of the insertion point
     */
    public static int positionOfGE( int index, List<Integer> data ) {
        int nextValue = data.get( index );
        int i = index;
        while( i > 0 && data.get(i - 1) > nextValue ) {
            i = i - 1;
        } // while
        return i;
    } // positionOfGE( int, List<Integer> )

    /**
     * Create a hole in a list into which to insert
     * an element in the next step of an insertion sort.
     *
     * @param i is the index of the element to be inserted
     * (0 <= j <= i <= data.size - 1)
     * @param j is the position at which the element is to
     * be inserted (0 <= j <= i <= data.size - 1)
     */
    public static void createHole( int i, int j, List<Integer> data ) {
        for( int k = i; k > j; k-- ) {
            data.set( k, data.get(k - 1) );
        } // for
        data.set( j, null );
//          while( i > j ) {
//              swap( i, i - 1, data );
//              i = i - 1;
//          } // while
    } // createHole( int, int, List<Integer> )

    /**
     * Sort a list using the insertion sort algorithm.
     *
     * @param data is the list to be sorted
     */
    public static void insertionSort( List<Integer> data ) {
     /* Insert the next value into the sorted part */
     /* of a list in an insertion sort. */

        for( int i = 1; i < data.size(); i++ ) {
```

```
            int j = positionOfGE( i, data );
            int temp = data.get( i );
            createHole( i, j, data );
            data.set( j, temp );
        } // for
    } // insertionSort( List<Integer> )

    /**
     * Merge the two parts of a list (when those
     * two parts are already sorted)
     *
     * The two parts of the list will contain
     * consecutive elements elements of the list.
     *
     * The two parts need not make up the whole
     * list---there may be other elements before and/or
     * after these two parts
     *
     * @param prefixStart is the index of the first
     * element in the first part of the list
     * @param suffixStart is the index of the first
     * element in the second part of the list
     * @param suffixEnd is the end of the list
     * element in the second part of the list
     * @param data is the list
     * @return the original list with the specified
     * two parts now sorted
     */
    public static List<Integer> merge( int prefixStart,
            int suffixStart, int suffixEnd, List<Integer> data ) {
        int i = prefixStart;
        int j = suffixStart;

        List<Integer> result = new ArrayList<>();

        while( i < suffixStart && j <= suffixEnd ) {
            if( data.get(i) < data.get(j) ) {
                result.add( data.get(i) );
                i++;
            } // if
            else {
                result.add( data.get(j) );
                j++;
            } // else
        } // while
```

```
        while( i < suffixStart ) {
            result.add( data.get(i) );
            i++;
        } // while

        while( j <= suffixEnd ) {
            result.add( data.get(j) );
            j++;
        } // while

        for( int k = 0; k < result.size(); k++ ) {
            data.set( k + prefixStart, result.get(k) );
        } // for

        return result;
    } // merge( List<Integer>, List<Integer> )

    /**
     * Sort a list using the merge sort algorithm.
     *
     * @param data is the list to be sorted.
     */
    public static void mergeSort( List<Integer> data ) {
    } // mergeSort( List<Integer> )

    public static void printList( List<Integer> list ) {
        for( int i = 0; i < list.size(); i++ ) {
            System.out.printf( "%3d ", list.get(i) );
            if( i != 0 && i % INTEGERS_ON_A_LINE == 0 ) {
                System.out.println();
            } // if
        } // for
        System.out.println();
    } // printList( List<Integer> )

    public static void main( String [] args ) {

        List<Integer> a = makePartlySortedList( 8, INTEGERS_ON_A_LINE, MAX_VALUE );
        List<Integer> b = makeSortedList( INTEGERS_ON_A_LINE, MAX_VALUE );
        List<Integer> c = makeUnsortedList( INTEGERS_ON_A_LINE, MAX_VALUE );

        System.out.println( "Partly sorted list." );
        printList( a );
        System.out.println();

        System.out.println( "Sorted list." );
```

```
printList( b );
System.out.println();

System.out.println( "After␣swapping␣elements␣3␣and␣5,␣the␣list␣is:"
swap( 3, 5, b );
printList( b );
System.out.println();

System.out.println( "Unsorted␣list." );
printList( c );
System.out.println();

System.out.println( "Value␣of␣smallest␣integer␣in␣list␣=␣" +
    minimum( c ) );
System.out.println();

System.out.println( "Position␣of␣smallest␣integer␣in␣list␣=␣" +
    positionOfMinimum( c ) );
System.out.println();

System.out.println( "Position␣of␣smallest␣integers␣in␣that␣\n" +
    "␣part␣of␣the␣list␣that␣begins␣at␣index␣8␣=␣" +
    positionOfMinimum( 8, c ) );
System.out.println();

System.out.println( "After␣sorting␣with␣the␣selection␣sort:␣" );
selectionSort( c );
printList( c );
System.out.println();

a = makePartlySortedList( 8, INTEGERS_ON_A_LINE, MAX_VALUE );
printList( a );
int j = positionOfGE( 8, a );
System.out.println( "The␣element␣at␣index␣8␣belongs␣at␣position␣" +
System.out.println();

System.out.println( "Here␣is␣the␣list␣with␣a␣hole␣created␣at␣the␣rig
createHole( 8, j, a );
printList( a );
System.out.println();

c = makeUnsortedList( INTEGERS_ON_A_LINE, MAX_VALUE );
System.out.println( "Here␣an␣unsorted␣list." );
printList( c );
insertionSort( c );
System.out.println( "And␣here␣is␣the␣same␣list␣after␣sorting␣with␣i
```

```
        printList( c );
        System.out.println();

        List<Integer> d = makeTwoPartList( 8, 8, MAX_VALUE );
        System.out.println( "Here is a list whose two parts are sorted." );
        printList( d );
        System.out.println();

        merge( 0, 8, d.size() - 1, d );
        System.out.println( "Here is the list after the merge of the two parts." );
        printList( d );

    } // main( String [] )

} // SortsSolution
```

Lesson 13

Classes as blueprints

13.1 Arithmetic with fractions

13.1.1 Greatest common divisor

A fraction can be reduced to its simplest form by dividing numerator and denominator by the greatest common divisor of both.

The greatest common divisor of two non-negative integers can be found with a recursive algorithm:

$$\begin{aligned} gcd(a, b) &= gcd(b, a \bmod b) \\ gcd(a, 0) &= a \end{aligned}$$

Example

$$\begin{aligned} gcd(144, 60) &= gcd(60, 144 \bmod 60) \\ &= gcd(60, 24) \\ &= gcd(24, 60 \bmod 24) \\ &= gcd(24, 12) \\ &= gcd(12, 24 \bmod 12) \\ &= gcd(12, 0) \\ &= 12 \end{aligned}$$

13.1.2 Addition

$$a = \frac{num_a}{den_a}$$

$$b = \frac{num_b}{den_b}$$

$$a + b = \frac{num_a \cdot den_b + num_b \cdot den_a}{den_a \cdot den_b}$$

Example

$$\begin{aligned}\frac{2}{5} + \frac{3}{8} &= \frac{2 \cdot 8 + 3 \cdot 5}{5 \cdot 8} \\ &= \frac{31}{40}\end{aligned}$$

13.1.3 Subtraction

$$a = \frac{num_a}{den_a}$$

$$b = \frac{num_b}{den_b}$$

$$a - b = \frac{num_a \cdot den_b - num_b \cdot den_a}{den_a \cdot den_b}$$

Example

$$\frac{2}{5} + \frac{3}{8} = \frac{2 \cdot 8 - 3 \cdot 5}{5 \cdot 8}$$

$$= \frac{1}{40}$$

13.1.4 Multiplication

$$a = \frac{num_a}{den_a}$$

$$b = \frac{num_b}{den_b}$$

$$a \times b = \frac{num_a \cdot num_b}{den_a \cdot den_b}$$

Example

$$\frac{2}{5} \times \frac{3}{8} = \frac{2 \cdot 3}{5 \cdot 8}$$

$$= \frac{6}{40}$$

$$= \frac{3}{20}$$

13.1.5 Division

$$a = \frac{num_a}{den_a}$$

$$b = \frac{num_b}{den_b}$$

$$a \div b = \frac{num_a \cdot den_b}{den_a \cdot num_b}$$

Example

$$\frac{3}{8} \div \frac{2}{5} = \frac{3 \cdot 5}{8 \cdot 2}$$

$$= \frac{15}{16}$$

Lesson 14

A Java class that models a fraction

14.1 Source code: Fraction.java

```
package apworkshop;

public class Fraction {
    private final int numerator;
    private final int denominator;

    public Fraction( int numerator, int denominator ) {
        int divisor = gcd( numerator, denominator );
        this.numerator = numerator/divisor;
        this.denominator = denominator/divisor;
    } // Fraction( int, int )

    public int getNumerator() {
        return this.numerator;
    } // getNumerator()

    public int getDenominator() {
        return this.denominator;
    } // getDenominator()

    public Fraction add( Fraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();
```

```
        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new Fraction( n0 * d1 + n1 * d0, d0 * d1 );
    } // add( Fraction )

    public Fraction subtract( Fraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new Fraction( n0 * d1 - n1 * d0, d0 * d1 );
    } // subtract( Fraction )

    public Fraction multiply( Fraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new Fraction( n0 * n1, d0 * d1 );
    } // multiply( Fraction )

    public Fraction divide( Fraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new Fraction( n0 * d1, d0 * n1 );
    } // divide( Fraction )

    @Override
    public String toString() {
        return this.getNumerator() + "/" + this.getDenominator();
    } // toString()

    private int gcd( int a, int b ) {
        if( b == 0 ) {
            return a;
        } // if
        else {
```

```
            return gcd( b, a % b );
        } // else
    } // gcd( int, int )

    public static void main( String [] args ) {
        Fraction a = new Fraction(2,5);
        Fraction b = new Fraction(3,8);

        System.out.println( "a = " + a );
        System.out.println( "b = " + b );

        System.out.println( "a + b = " + a.add(b) );
        System.out.println( "a - b = " + a.subtract(b) );
        System.out.println( "a * b = " + a.multiply(b) );
        System.out.println( "b / a = " + b.divide(a) );
    } // main( String [] )

} // Fraction
```

Lesson 15

Sorting Fractions

15.1 Source code: ComparableFraction.java

```
package apworkshop;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class ComparableFraction
      implements Comparable<ComparableFraction> {

    private static final int NUMBER_OF_FRACTIONS = 12;

    private final int numerator;
    private final int denominator;

    public ComparableFraction( int numerator, int denominator ) {
        int divisor = gcd( numerator, denominator );
        this.numerator = numerator/divisor;
        this.denominator = denominator/divisor;
    } // ComparableFraction( int, int )

    public int getNumerator() {
        return this.numerator;
    } // getNumerator()

    public int getDenominator() {
        return this.denominator;
```

```
    } // getDenominator()

    public ComparableFraction add( ComparableFraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new ComparableFraction( n0 * d1 + n1 * d0, d0 * d1 );
    } // add( ComparableFraction )

    public ComparableFraction subtract( ComparableFraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new ComparableFraction( n0 * d1 - n1 * d0, d0 * d1 );
    } // subtract( ComparableFraction )

    public ComparableFraction multiply( ComparableFraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new ComparableFraction( n0 * n1, d0 * d1 );
    } // multiply( ComparableFraction )

    public ComparableFraction divide( ComparableFraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();

        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        return new ComparableFraction( n0 * d1, d0 * n1 );
    } // divide( ComparableFraction )

    @Override
    public String toString() {
        return this.getNumerator() + "/" + this.getDenominator();
    } // toString()
```

```
    private int gcd( int a, int b ) {
        if( b == 0 ) {
            return a;
        } // if
        else {
            return gcd( b, a % b );
        } // else
    } // gcd( int, int )

    @Override
    public int compareTo( ComparableFraction otherFraction ) {
        int n0 = this.getNumerator();
        int d0 = this.getDenominator();
        int n1 = otherFraction.getNumerator();
        int d1 = otherFraction.getDenominator();

        int product = (d1 * n0) / (n1 * d0);

        if( (d1 * n0)/(n1 * d0) > 1 ) {
            return +1;
        } // if
        else if( product < 1 ) {
            return -1;
        } // else if
        else {
            return 0;
        } // else
    } // compareTo( ComparableFraction )

    public static void main( String [] args ) {

        Random rng = new Random();
        List<ComparableFraction> fractions = new ArrayList<>();

        for( int i = 0; i < NUMBER_OF_FRACTIONS; i++ ) {
            int denominator = 1 + rng.nextInt( 10 );
            int numerator = 1 + rng.nextInt( denominator );
            ComparableFraction f = new ComparableFraction( numerator, denominator );
            fractions.add( f );
        } // for

        System.out.println( "Unsorted list of fractions: " );
        for( ComparableFraction f : fractions ) {
            System.out.println( "\t" + f );
        } // for
```

```
        System.out.println();

        Collections.sort( fractions );

        System.out.println( "Sorted list of fractions: " );
        for( ComparableFraction f : fractions ) {
            System.out.println( "\t" + f );
        } // for

    } // main( String [] )

} // ComparableFraction
```

Lesson 16

Review

1. Here are five steps for writing a function. Which step should be last?
   (a) Identify the type of value that the function will return to its caller.
   (b) Describe the purpose of the function in a single, plainly worded, simple sentence.
   (c) List the number and types of parameters (the information that the function needs from its caller).
   (d) Compose the sequence of logical and arithmetic operations that will produce the value to be returned to the caller.
   (e) Give the function a meaningful name.

Answer:

(d) Compose the sequence of logical and arithmetic operations that will produce the value to be returned to the caller.

2. Here is a trace of a selection sort. What will be the next row in this trace? That is, what will the order of the elements in the list be at Step #5?

Step								
0	54	78	34	93	40	53	97	50
1	**34**	78	54	93	40	53	97	50
2	**34**	**40**	54	93	78	53	97	50
3	**34**	**40**	**50**	93	78	53	97	54
4	**34**	**40**	**50**	**53**	78	93	97	54

Answer:

Step								
0	54	78	34	93	40	53	97	50
1	**34**	78	54	93	40	53	97	50
2	**34**	**40**	54	93	78	53	97	50
3	**34**	**40**	**50**	93	78	53	97	54
4	**34**	**40**	**50**	**53**	78	93	97	54
5	**34**	**40**	**50**	**53**	**54**	93	97	78
6	**34**	**40**	**50**	**53**	**54**	**78**	97	93
7	**34**	**40**	**50**	**53**	**54**	**78**	**93**	97
8	**34**	**40**	**50**	**53**	**54**	**78**	**93**	**97**

3. Here is a trace of an insertion sort. What will be the next row in this trace? That is, what will the order of the elements in the list be at Step #4?

Step								
0	**72**	34	35	00	30	84	87	77
1	**34**	**72**	35	00	30	84	87	77
2	**34**	**35**	**72**	00	30	84	87	77
3	**00**	**34**	**35**	**72**	30	84	87	77

Answer:

Step								
0	**72**	34	35	00	30	84	87	77
1	**34**	**72**	35	00	30	84	87	77
2	**34**	**35**	**72**	00	30	84	87	77
3	**00**	**34**	**35**	**72**	30	84	87	77
4	**00**	**30**	**34**	**35**	**72**	84	87	77
5	**00**	**30**	**34**	**35**	**72**	**84**	87	77
6	**00**	**30**	**34**	**35**	**72**	**84**	**87**	77
7	**00**	**30**	**34**	**35**	**72**	**77**	**84**	**87**
8	**00**	**30**	**34**	**35**	**72**	**77**	**84**	**87**

4. One way to measure the amount of work that a sorting algorithm does is by counting the number of times that it compares one element of a list to another.

You can see from the tables shown in the previous questions that the number of comparisons is...

$$1 + 2 + 3 + 4 + \cdots + (N - 2) + (N - 1)$$

For very large values of N, which expression most closely approximates the value of this sum:

(a) $\log N$
(b) N
(c) $N \log N$
(d) N^2

Answer:

(d) N^2

5. This function is supposed to move the integer at position i in an array to position j and move the integer at position j to position i.

   Two more lines of code are needed to complete the definition of this function.

   Supply the missing code where indicated.

```
void swap( int *data, int i, int j ) {
    // WRITE A STATEMENT HERE
    data[i] = data[j];
    // WRITE A STATEMENT HERE
} // swap( int *, int, int )
```

Answer:

```
void swap( int *data, int i, int j ) {
    int temp = data[i];
    data[i] = data[j];
    data[j] = temp;
} // swap( int *, int, int )
```

6. Here is a function that finds the smallest integer in an array. Change the function so that it returns the index of the smallest integer instead of the value of the smallest integer.

```
int findMin (int *data, int size ){
    int bestGuessSoFar = data[0];

    for( int i = 1; i < size; i++ ) {
        if( data[i] < bestGuessSoFar ) {
            bestGuessSoFar = data[i];
        } // if
    } // for

    return bestGuessSoFar;
} // findMin( int *, int )
```

Answer:

```
int findPositionOfMin (int *data, int size ){
    int bestGuessSoFar = 0;

    for( int i = 1; i < size; i++ ) {
        if( data[i] < data[bestGuessSoFar] ) {
            bestGuessSoFar = i;
        } // if
    } // for

    return bestGuessSoFar;
} // findPositionOfMin( int *, int )
```

7. This function sorts a list using the selection sort algorithm.

   What does the call to findPosOfMinPart() accomplish in this function?

```
void selectionSort( int *data, int size ) {
    for( int i = 0; i < size; i++ ) {
        int j = findPosOfMinInPart( data, size, i );
        swap( data, i, j );
    } // for
} // selectionSort( int *, int )
```

(a) it finds the position of the smallest integer in that part of the array named data whose first element has index 0 and whose last element has index i

(b) it finds the position of the smallest integer in that part of the array named data whose first element has index i and whose last element has index $size - 1$

(c) it finds the position of the smallest integer among the first i elements of the array named *data*

(d) it finds the position of the smallest integer among the last i elements of the array named *data*

Answer:

(b) it finds the position of the smallest integer in that part of the array named data whose first element has index i and whose last element has index $size - 1$

8. What does this function accomplish?

```
int positionOfGE( int index, int *data ) {
    int nextValue = data[ index ];
    int i = index;
    while( i > 0 && data[ i - 1 ] > nextValue ) {
        i = i - 1;
    } // while
    return i;
} // positionOfGE( int, int * )
```

(a) searches from left to right to find the index of the last integer in the array named *data* that is less than or equal to *data*[*index*]

(b) searches from right to left to find the index of the last integer in the array named *data* that is greater than *data*[*index*]

(c) searches from left to right to find the index of the first integer in the array named *data* that is less than or equal to *data*[*index*]

(d) searches from right to left to find the index of the first integer in the array named *data* that is greater than *data*[*index*]

Answer:

(b) searches from right to left to find the index of the last integer in the array named *data* that is greater than *data*[*index*]

9. The following example includes a recursive function. Which features are common to all recursive functions?

   (a) The main() function does not call the recursive function directly.
   (b) The function includes a call to itself.
   (c) The function includes an **if** statement.
   (d) The type of value that the function returns to its caller matches the types of its parameters.

```
package apworkshop;

public class NewtonsMethod {

    private static final double EPSILON = 1E-8;

    private static double squareRootHelper( double x, double guess ) {
        if( Math.abs(x - guess * guess) < EPSILON ) {
            return guess;
        } // if
        else {
            return squareRootHelper( x, (guess + x/guess)/2 );
        } // else
    } // squareRootHelper( double, double )

    public static double squareRoot( double x ) {
        return squareRootHelper( x, 1.0 );
    } // squareRoot( double )

    public static void main( String [] args ) {
        double x = Double.parseDouble( args[0] );

        System.out.printf( "Square root of %8.4f = %8.4f\n",
            x, squareRoot(x) );
    } // main( String [] )

} // NewtonsMethod
```

Answer:

(a) (incorrect)
(b) (correct) The function includes a call to itself.
(c) (correct) The function includes an **if** statement.
(d) (incorrect)

10. At the heart of the merge sort is a function that merges two sublists that are already sorted. Here is the first part of that function.

    When does execution of the **while** loop end?

    (a) The execution of the loop stops when the function has copied all values from *data* to *result*.
    (b) The execution of the loop stops when the function has copied all values from both the first sublist (*prefix*) **and** the second sublist (*suffix*) to the *result* array.
    (c) The execution of the loop stops when the function has copied all values from either the first sublist (*prefix*) **or** the second sublist (*suffix*) to the *result* array.
    (d) The execution of the loop stops when the function finds a value in the first sublist (*prefix*) that is greater than some value in the second sublist (*suffix*).

```
int i = prefixStart;
int j = suffixStart;
int k = 0;

int subsetSize = suffixEnd - prefixStart + 1;

int [] result = new int[ subsetSize ];

while( i < suffixStart && j <= suffixEnd ) {
    if( data[i] < data[j] ) {
        result[k] = data[i];
        i++;
    } // if
    else {
        result[k] = data[j];
        j++;
```

```
    } // else
    k++;
} // while
```

Answer:

(a) (incorrect)

(b) (incorrect)

(c) (correct) The execution of the loop stops when the function has copied all values from either the first sublist (*prefix*) **or** the second sublist (*suffix*) to the *result* array.

(d) (incorrect)

11. The merge() function also contains these two additional **while** loops.

    Assuming that neither of the sublists that the function is given are empty.

    How many of these two loops might run in an execution of the merge() function?

    (a) 0 or 1

    (b) 0 or 2

    (c) 1 or 2

    (d) 1

```
while( i < suffixStart ) {
    result[k] = data[i];
    i++;
    k++;
} // while

 while( j <= suffixEnd ) {
    result[k] =  data[j];
    j++;
    k++;
} // while
```

Answer:

(a) (incorrect)
(b) (incorrect)
(c) (incorrect)
(d) (correct) 1

12. The selection sort searches from left to right through the unsorted part of a list.

    The insertion sort searches from right to left through the sorted part of a list.

    Why is the insertion sort often faster than the selection sort?

    (a) The sorted part of the list is shorter than the unsorted part.
    (b) The minimum value can be found more quickly in the sorted part of the list.
    (c) It can stop as soon as it finds a value that is not greater than the value it is inserting into the sorted part of the list.
    (d) It can compare one value to two other values at the same time.

Answer:

(a) (incorrect)
(b) (incorrect)
(c) (correct) It can stop as soon as it finds a value that is not greater than the value it is inserting into the sorted part of the list.
(d) (incorrect)

13. Here is a proof by induction. How is it like a recursive algorithm?

    (a) It has at least one base case.
    (b) It is a statement about very large numbers.
    (c) It specifies different actions for odd and even inputs.

(d) It has a boolean return value.

Statement: “The number of people who have shaken hands with an odd number of people is even.

Proof: Before the invention of handshakes, zero people had shaken hands with an odd number of people. Since zero is an even number, the statement was once true.

After the first handshake, two (an even number) people had each shaken hands with one (an odd number) other person. Again, the statement was true.

There are only 2 kinds of people. Members of the ODD set have shaken hands with an odd number of people. Members of the EVEN set have shaken hands with an even number of people.

There are only 3 kinds of handshakes.

- An ODD person can shake hands with an ODD person. Both people leave the ODD set (an odd number plus one is even).

  If the size of the ODD set was even before, it is still even after their departure, since an even number minus two is even.
- An EVEN person can shake hands with an EVEN person. Both people join the ODD set (an even number plus one is odd).

  If size of the ODD set was even before, it is even after the addition of two more people.
- An ODD person can shake hands with an EVEN person. One person leaves the ODD set and one person joins the ODD set.

  The size of the ODD set remains unchanged. If it was even, it remains even.

The number of people who had shaken hands with an even number of people was once even. No handshake since the invention of handshakes could have changed that fact. QED.

Answer:

(a) (correct) It has at least one base case.
(b) (incorrect)
(c) (incorrect)
(d) (incorrect)

14. Our employer as asked us to write a function that computes $g(x, y)$:

$$g(x, y) = \frac{(y_3 - y_2)x + (x_2 - x_3)y + (x_3y_2 - x_2y_3)}{(y_3 - y_2)x_0 + (x_2 - x_3)y_0 + (x_3y_2 - x_2y_3)}$$

Our teammates have already written these two functions for us:

```
double numerator( double x, double y, double x0, double y0,
       double x2, double y2, double x3, double y3 ) {

    return (y3 - y2) * x + (x2 - x3) * y + (x3 * y2 - x2 * y3);
} // numerator()

double denominator( double x0, double y0,
       double x2, double y2, double x3, double y3 ) {

    return (y3 - y2) * x0 + (x2 - x3) * y0 + (x3 * y2 - x2 * y3);
} // denominator()
```

How should we proceed?

(a) Write our own clean code.
(b) Copy the expressions from our teammates' code into our code everywhere we need to compute $g(x, y)$.
(c) Call our teammate's two functions everywhere we need to compute $g(x, y)$.
(d) Define a third function that calls our teammates' two functions.

Answer:

(a) (incorrect)
(b) (incorrect)
(c) (incorrect)
(d) (correct) Define a third function that calls our teammates' two functions.

15. What is the post office problem?

    (a) It is an effort to route mail efficiently through the postal network.
    (b) It is an effort to draw boundaries around the districts that each post office in a region should serve.
    (c) It is an effort to share work fairly among several post offices.
    (d) It is an effort to manage waiting lines inside post offices.

Answer:

(a) (incorrect)
(b) (correct) It is an effort to draw boundaries around the districts that each post office in a region should serve.
(c) (incorrect)
(d) (incorrect)

16. You have seen the merge sort, Euclid's algorithm for computing greatest common divisors, and an algorithm for raising a number to a non-negative integer power. They were all examples of "divide-and-conquer" algorithms.

    What is a characteristic of a "divide-and-conquer" algorithm?

    (a) It use loops to build the solution of a problem one small step at a time.
    (b) It expresses the solution of a large problem in terms of the solution of smaller problems.
    (c) It generates a guess, tests the accuracy of the guess, and then produces an improved guess.
    (d) It calls several small functions rather than one large function.

Answer:

(a) (incorrect)

(b) (correct) It expresses the solution of a large problem in terms of the solution of smaller problems.

(c) (incorrect)

(d) (incorrect)

17. The GraphicsContext class has methods that can be used to draw pictures. You used the methods of this class to draw your landscape.

    Suppose that you have an instance of the GraphicsContext class named gc.

    (a) Show a call to a method of the GraphicsContext class that will alter some drawing attribute (for example, the color or width of lines). (Your answer need not be perfect.)

    (b) Show a call to a method of the GraphicsContext class that draws something on the screen.

Answer:

(a) gc.setStroke(Color.BLACK);

(b)

```
double[] xCoordinates = new double[3];
double[] yCoordinates = new double[3];

xCoordinates[0] = 0.0;
xCoordinates[1] = 100.0;
xCoordinates[2] = 100.0;

yCoordinates[0] = 0.0;
yCoordinates[1] = 0.0;
yCoordinates[2] = 100.0;

gc.strokePolygon(xCoordinates, yCoordinates, 3);
```

18. A Java class may extend another class or it may implement an interface.

    An interface, like a class, lists the names of methods, the types of values that those methods return to their callers, and the numbers and types of parameters that those methods require. Unlike a class, an interface does not define how a method will produce its result.

    Why did the designers of the Java programming language choose to make a distinction between classes and interfaces?

    Answer:

    If Java allowed a class to extend two classes, it might happen that the class would inherit methods from both classes that had the same name and same number and types of parameters. These methods might work in different ways and produce different results. The interpreter could not know which version to execute.

    If a class implements two interfaces, there is no conflict, because, although both interfaces might specify methods with the same name, neither defines the method—the relationship between class and interface requires the author of the class to give the method a single, unambiguous definition.

19. In each of the classes that we wrote we included a piece that looked like a method in most ways. It differed from the methods by the fact that it lacked a return type and a return statement. It also differed from the methods by the fact that its name matched the name of the class.

    What do we call this part of a class?

    Answer:

    It is a constructor.

20. Encapsulation is a key goal of object-oriented programming. Support for encapsulation is a key feature of object-oriented languages.

In the Java programming language, this support takes the form of instance variables, accessor (getter) methods, mutator (setter) methods, and **private** and **public** qualifiers.

Why is encapsulation a good idea?

Answer:

Encapsulation hides the details of how a class organizes, combines, and changes data. These allows members of a software engineering team to each focus their efforts on their own parts of a project. They are relieved from the need to know all of the details of their teammates' contributions. Encapsulation prevents a person who is responsible for one part of a program from writing code that accidentally makes changes in another part. Encapsulation makes programming easier and safer.

21. What is a stub method? Why might you write a stub method?

Answer:

A stub method defines the name of the method, the type of value it returns to its caller, and the number and types of its parameters. The body of stub method contains only a **return** statement. It returns an arbitrary value (for example, 0, 1, **true**, or **false**). This will most often be an incorrect result. However, with the stub method in place, the program will compile and run, and the programmer can work on other parts of the program until the programmer finds the knowledge needed to complete the definition of the method.

22. What is the value of regression testing?

Answer:

Software engineers continue running tests even after the code passes the tests. They do this because they understand that defects, once repaired,

can reappear. If one programmer misunderstood specifications, made an unjustified assumption about how clients will use the code, or got the logic or arithmetic wrong, another programmer might repeat the mistake at a later time. Code evolves, many people touch the code, and changes in one part of a program can affect other, distant parts of the program. Regression testing—repeated, continued testing—protects against the reintroduction of errors.

23. The availability of components makes the design and manufacture of many kinds of products easier. Engineers do not have to design and build every part of product when they have the option of using components that someone else has already designed, built, and tested.

    What is the form of the components that software engineers create and share? (That is, what is a word that describes a component of a software program?)

Answer:

A class is the must fundamental component of an object-oriented program. It is a shareable and reusable component.

24. Inheritance is another key concept in the object-oriented discipline. How does the use of inheritance enable a programmer to solve a problem with less work than might otherwise be necessary?

Answer:

When one class extends another class, it inherits methods and variables from the parent class. The author of the class that inherits does not have to repeat the definitions of these elements.

Inheritance is an "is a" (or "is a kind of") relationship. The class that inherits models a special kind of the thing that the parent class models. The author of the subclass has only to define that features that distinguish the subclass from the parent class.

25. Here is a function that finds the smallest integer in an array of integers.

```
int minimum( int [] data ) {
    int bestGuessSoFar = data[0];
    for( int i = 1; i < data.length; i++ ) {
        if( data[i] < bestGuessSoFar ) {
            bestGuessSoFar = data[i];
        } // if
    } // for
    return bestGuessSoFar;
} // minimum( int [] )
```

A function that finds the position of the smallest integer in an array of integers is very similar. It differs in only three places.

```
int positionOfMinimum( int [] data ) {
    int bestGuessSoFar = ????;
    for( int i = 1; i < data.length; i++ ) {
        if( data[i] < ???? ) {
            bestGuessSoFar = ????;
        } // if
    } // for
    return bestGuessSoFar;
} // positionOfMinimum( int [] )
```

(a) This function will return an index. It begins by guessing that index is equal to what?

(This replaces the first set of question marks.)

(b) This function goes on to compare the values at every position in the array with the value at the position where it has guessed that the smallest value is.

In the **if** statement, it compares data[i] with what?

(This replaces the second set of question marks.)

(c) When the function finds the position of a smaller integer it updates its guess of the position of the smallest element of the array.

What belongs on the right-hand side of the assignment statement within the **if** statement?

(This replaces the third set of question marks.)

Answer:

(a) **int** bestGuessSoFar = 0;

(b) **if**(data[i] < data[bestGuessSoFar]) {

(c) bestGuessSoFar = i;

26. A logarithm appears in the expressions that describe how much work must be done in a binary search or in sorting a list with the quicksort or merge sort algorithms.

    In your high school algebra class, you learned that a logarithm is the power to which we have to raise one number to get a given value. For example, the logarithm base 2 of 16 is 4 because $2^4 = 16$.

    A different but equivalent definition of logarithm explains why logarithms show up in the analysis of divide-and-conquer algorithms like binary search, quicksort, and merge sort.

    Here is an application of that different definition.

$$512/2 = 256 \quad (1)$$
$$256/2 = 128 \quad (2)$$
$$128/2 = 64 \quad (3)$$
$$64/2 = 32 \quad (4)$$
$$32/2 = 16 \quad (5)$$
$$16/2 = 8 \quad (6)$$
$$8/2 = 4 \quad (7)$$
$$4/2 = 2 \quad (8)$$
$$2/2 = 1 \quad (9)$$

$$\log_2 512 = 9$$

Put this definition into words: "The logarithm base 2 of a positive power of 2 is..."

Answer:

The logarithm base 2 of a positive power of 2 is the number of times we have to divide the number by 2 before reaching 1.

A divide by conquer algorithm cuts a problem into half again and again, until it gets down to a trivial problem. It expresses the solution of big problems as the solution of a smaller problems.

27. The selection sort algorithm is $O(N^2)$. How much longer would you expect to wait for a program that uses this algorithm to put two million values into order, as compared to the time required to sort one million objects?

Answer:

Two million is twice one million. A $O(N^2)$ algorithm takes four times as long to produce an answer when the size of the input is doubled.

28. Here are the first few steps of a selection sort. What is the next line in this table?

16	78	82	61	21	96	55	42	05	40	49	44
05	78	82	61	21	96	55	42	16	40	49	44
05	**16**	82	61	21	96	55	42	78	40	49	44
05	**16**	**21**	61	82	96	55	42	78	40	49	44
05	**16**	**21**	**40**	82	96	55	42	78	61	49	44
05	**16**	**21**	**40**	**42**	96	55	82	78	61	49	44

Answer:

05	**16**	**21**	**40**	**42**	**44**	55	82	78	61	49	96

29. (a) Which sorting algorithm needs memory sufficient to hold two copies of the list to be sorted?
    (b) Which sorting algorithm repeatedly searches from right to left through the already sorted part of the list to find the position to which the first element in the unsorted part of the list should be moved?
    (c) Which sorting algorithm repeatedly searches from left to right through the still unsorted part of the list to find the position of the element with the smallest value?

Answer:

(a) merge sort
(b) insertion sort
(c) selection sort

30. Recall that...

- $2^x \times 2^x = 2^{2x}$
- $2^y/2^x = 2^{y-x}$
- $\log_2 2^x = x$
- $2^{10} \approx$ one thousand
- $2^{20} \approx$ one million
- $2^{30} \approx$ one billion
- $2^{40} \approx$ one trillion
- one year $\approx$ 30 million seconds

Now, suppose that...

- you have a list of 2^{30} items to sort
- your computer can execute 2^{30} instructions per second

(a) How long will it take to complete the job with an algorithm that requires the execution of N^2 instructions?
(b) How long will it take to complete the job with an algorithm that requires the execution of $N \log_2 N$ instructions?

Answer:

(a) $2^{30} \times 2^{30} = 2^{60} \approx$ one million trillion instructions. 2^{60} instructions $/2^{30}$ instructions per second $= 2^{30} =$ one billion seconds. One billion seconds $\approx$ 30 years.

(b) $2^{30} \times 30 \approx$ thirty billion instructions. $30 \cdot 2^{30}$ instructions $/2^{30}$ instructions per second = 30 seconds.

31. Complete this stub method.

    - Find the index of the smallest integer in that part of a list of integers that begins at a specified position
    - *startingIndex* is the position of that part of the list in which to begin the search.
    - *data* is the list in which to search.
    - The method returns the index of the smallest integer in that part the list that begins the specified position.

```
public static int positionOfMinimum( int startingIndex,
        List<Integer> data ) {

    return 0;
} // positionOfMinimum( int, List<Integer> )
```

Answer:

```
public static int positionOfMinimum( int startingIndex,
        List<Integer> data ) {

    int bestGuessSoFar = startingIndex;
    for( int i = startingIndex + 1; i < data.size(); i++ ) {
        if( data.get(i) < data.get(bestGuessSoFar) ) {
            bestGuessSoFar = i;
        } // if
    } // for
    return bestGuessSoFar;
} // positionOfMinimum( int, List<Integer> )
```

32. Complete this stub method.

    - This method exchanges the values of the elements at positions i and j in a list.
    - *i* is an index in the list and in the interval [0, data.size() - 1].

- *j* is an index in the list and in the interval [0, data.size() - 1].
- *data* is the list that contains the two elements.

```
public static void swap( int i, int j,
        List<Integer> data ) {

} // swap( int, int, List<Integer> )
```

Answer:

```
public static void swap( int i, int j,
        List<Integer> data ) {

    int temp = data.get(i);
    data.set(i, data.get(j) );
    data.set(j, temp );
} // swap( int, int, List<Integer> )
```

33. Complete this stub method.

- This method sorts a list of integers using the selection sort algorithm.
- *data* is the list to be sorted.

```
public static void selectionSort( List<Integer> data ) {
    for( int i = 0; i < data.size(); i++ ) {

      // TO-DO: Add calls to positionOfMinimum()
      // and swap() here.

    } // for
} // selectionSort( List<Integer> )
```

Answer:

```
public static void selectionSort( List<Integer> data ) {
    for( int i = 0; i < data.size(); i++ ) {
        int j = positionOfMinimum( i, data );
        swap( i, j, data );
    } // for
} // selectionSort( List<Integer> )
```

34. Complete this stub method by writing the second part of the condition for the continuation of the looping.

    - This method finds the position in the sorted part of a list in which to insert the next element.
    - This will be the last element (in a search from right to left) that is greater than or equal to a given element.
    - *index* is the position at which to begin the left to right search and is also the position of the element to which all other elements in the search will be compared.
    - *data* is the list in which to search.
    - This method returns the index of the insertion point.

```
public static int positionOfGE( int index, List<Integer> data ) {
    int nextValue = data.get( index );
    int i = index;
    while( i > 0 /* TO-DO: Write second part of condition. */ ) {
        i = i - 1;
    } // while
    return i;
} // positionOfGE( int, List<Integer> )
```

Answer:

```
public static int positionOfGE( int index, List<Integer> data ) {
    int nextValue = data.get( index );
    int i = index;
    while( i > 0 && data.get(i - 1) > nextValue ) {
        i = i - 1;
    } // while
    return i;
} // positionOfGE( int, List<Integer> )
```

35. Complete this stub method.

    - This method creates a hole in a list into which to insert an element in the next step of an insertion sort.
    - i is the index of the element to be inserted ($0 <= j <= i <= data.size - 1$).
    - j is the position at which the element is to be inserted ($0 <= j <= i <= data.size - 1$).

```
public static void createHole( int i, int j,
        List<Integer> data ) {

    for( int k = i; k > j; k-- ) {

      // TO-DO: Assign a new value to the k-th element of data.

    } // for
    data.set( j, null );
} // createHole( int, int, List<Integer> )
```

Answer:

```
public static void createHole( int i, int j,
        List<Integer> data ) {

    for( int k = i; k > j; k-- ) {
        data.set( k, data.get(k - 1) );
    } // for
    data.set( j, null );
} // createHole( int, int, List<Integer> )
```

36. Complete this stub method with these statements:

    (a) data.set(j, temp);

    (b) createHole(i, j, data);

    (c) **int** j = positionOfGE(i, data);

(d) **int** temp = data.get(i);

- This method sorts a list using the insertion sort algorithm.
- *data* is the list to be sorted.

```
public static void insertionSort( List<Integer> data ) {
    for( int i = 1; i < data.size(); i++ ) {

        // TO-DO: Place 4 statements in the right order here.

    } // for
} // insertionSort( List<Integer> )
```

Answer:

```
public static void insertionSort( List<Integer> data ) {
    for( int i = 1; i < data.size(); i++ ) {
        int j = positionOfGE( i, data );
        int temp = data.get( i );
        createHole( i, j, data );
        data.set( j, temp );
    } // for
} // insertionSort( List<Integer> )
```

37. Complete this stub method.

    - Merge the two parts of a list (when those two parts are already sorted).
    - The two parts of the list will contain consecutive elements elements of the list.
    - The two parts need not make up the whole list—there may be other elements before and/or after these two parts.
    - *prefixStart* is the index of the first element in the first part of the list.
    - *suffixStart* is the index of the first element in the second part of the list.
    - *suffixEnd* is the end of the list element in the second part of the list.
    - *data* is the list.

- This method returns the original list with the specified two parts now sorted.

```
public static void merge( int prefixStart,
        int suffixStart, int suffixEnd, List<Integer> data ) {
    int i = prefixStart;
    int j = suffixStart;

    List<Integer> result = new ArrayList<>();

    while( i < suffixStart && j <= suffixEnd ) {
        if( data.get(i) < data.get(j) ) {
            result.add( data.get(i) );
            i++;
        } // if
        else {
            result.add( data.get(j) );
            j++;
        } // else
    } // while

    // TO-DO: Write two more while loops here.

    for( int k = 0; k < result.size(); k++ ) {
        data.set( k + prefixStart, result.get(k) );
    } // for

} // merge( int, int, int, List<Integer> )
```

Answer:

```
public static void merge( int prefixStart,
        int suffixStart, int suffixEnd, List<Integer> data ) {
    int i = prefixStart;
    int j = suffixStart;

    List<Integer> result = new ArrayList<>();

    while( i < suffixStart && j <= suffixEnd ) {
        if( data.get(i) < data.get(j) ) {
            result.add( data.get(i) );
```

```
            i++;
        } // if
        else {
            result.add( data.get(j) );
            j++;
        } // else
    } // while

    while( i < suffixStart ) {
        result.add( data.get(i) );
        i++;
    } // while

    while( j <= suffixEnd ) {
        result.add( data.get(j) );
        j++;
    } // while

    for( int k = 0; k < result.size(); k++ ) {
        data.set( k + prefixStart, result.get(k) );
    } // for

} // merge( int, int, int, List<Integer> )
```

38. When does the recursion stop in the execution of this method?

```
public static void mergeSortHelper( int i, int j,
        List<Integer> data ) {
    if( i == j ) {
    } // if
    else if( j - i == 1 ) {
        merge( i, j, j, data );
    } // else if
    else {
        int k = (i + j)/2;
        mergeSortHelper( i, k - 1, data );
        mergeSortHelper( k, j, data );
        merge( i, k, j, data );
    } // if
} // mergeSortHelper( int, int, List<Integer> )
```

Answer:

The method repeatedly divides a list into two smaller lists. It then calls itself, first with one of the sublists, and then with the other. When these two recursive calls return, the method merges the two (now sorted) sublists.

Recursion stops when the divisions result in a sublist that has two (or fewer) elements.

Lesson 17

What we have learned and what comes next

- post office problem, Java programming, and linear algebra
  - modeled random numbers, vectors, points, colors, and queues in C
  - abstract data types
  - dynamic memory allocation and pointers
  - used vectors to model RGB colors
  - used vectors to model points in two-dimensional space
  - used a matrix and matrix-vector multiplication to model a rotation
  - used matrices and vectors to model a system of linear equations
  - solved system of equations with an inverse matrix
  - solution of system gave us description of a line and a function the computes distance of a point from the line
- what we learned about how to solve problems
  - write for 2 audiences: the machine and other people
  - take small steps
  - test at every stage
  - design simple test cases
  - know what answer you expect before running tests
  - offer help to others, seek help from others
  - everyone gets stuck sometimes! you have the talent!
  - even when working alone, think of yourself as part of a team

  - your software might have a long life and be used by many people!

- why use the Java programming language?
  - widely available, universally known
  - lots of great code written in Java, available for use
  - language in which Unix is written
  - great tools for working with Java programs
  - efficient computation
  - for example, no checks on array bounds, no garbage collection
  - can mix assembly language with C (for example, to load values in registers or specific memory locations)
  - can indicate that a variable should be stored in a register
  - can use system calls directly to. . .
    * allocate and deallocate memory
    * create, stop, suspend, communicate among processes/threads
    * open, close, read, and write files (or connections to network)

- why not use the Java programming language?
  - dangerous!
    * we are responsible for freeing the memory we allocate
    * possibility of dangling pointers, memory leaks
  - hard to read and write
    * pointers and pointers to pointers

- what might we hope to see in a better language?
  - garbage collection—automatic recycling of blocks of memory that we allocated but no longer need
  - check for exceptions (such as in index for an array that is out of bounds) and give us a means to respond
  - package pieces of our code in a form that makes sharing more convenient
  - a means to write functions that can accept functions as parameters and return new functions to their callers
  - a means to define a specialized version of an abstract data type that we have already written without having to rewrite the parts that the specialized version will borrow from the more general version
  - better ways of examining each element in a collection

- what is next?

- queue
  - First In / First Out is FIFO
  - can be built on a linked list
  - used to model waiting lines (customers in a bank, traffic on a street, processes scheduled for execution in a computer)
- stack
  - Last In / First Out is LIFO
  - can be build on a linked list
  - used to hold parameters, return addresses, and return values of function calls
  - used for evaluation of arithmetic expressions
- priority queue
  - a tree
  - each node has a value less than values of its children
  - no left/right ordering of values in trees
  - build on a "heap" (careful: "heap" is also used to mean something else in computer science!)
  - an array—move elements within array using clever arithmetic on indices
  - during insertions and deletions, elements sink to bottom of tree and float to top of tree
- binary search tree
  - value of left child less than value at parent node
  - value of right child more than value at parent node
  - nodes like those in doubly linked list (labeled left and right instead of previous and next)
- balanced trees
  - AVL trees
  - 2–3 trees
  - red–black trees
- hash tables
- what to learn about these data structures
  - common uses
  - how to build them (do it just once, for education)
  - how to use them through popular libraries of software (someone else designed and built very good versions of these abstract data types)