**Chapter 2**

**Principles of Programming and Software Engineering**

**1** /\*\* findGCD – Find the greatest common divisor of two integers.

 \* This method takes two positive integer parameters, and finds the

 \* largest positive integer that divides into both without remainders.

 \* For example, the GCD of 42 and 30 is 6.

 \* Precondition: a >= 1 and b >= 1.

 \* Postcondition: a % gcd == 0 and

 \* b % gcd == 0 and

 \* for all x where a % x == 0 and b % x == 0,

 \* gcd is the largest such x.

 \*/

**2** Use a class Money that stores the number of dollars and cents as private data members. When declaring new objects or changing the data of existing ones, the class methods should ensure that if the number of cents exceeds 99, the number of dollars and cents are changed to represent the monetary amount in its most reduced form.

public static main (String args[])

{

Money itemCost = new Money(dollars, cents);

Money amountPaid = new Money(d, c);

Money change = getChange(itemCost, amountPaid);

change.display();

 }

public static Money getChange(Money price, Money payment)

 // ----------------------------------------------------------------

 // Computes the change remaining from purchasing an item costing

 // price when paying with payment.

 // Preconditions: price and payment are objects of class Money and

 // have been initialized to their desired values.

 // Postconditions: returns an object of class Money representing the

 // change to be received. If price < payment, the amount owed is

// returned as a negative value.

 // ----------------------------------------------------------------

**3a** /\*\* incrementHour – add 1 hour to a Time object.

 **\*** The question asks us to increment the time by one day. But since we only

 \* store hours, minutes, and seconds, changing the day would have no effect.

 \* Instead, let’s consider the problem of advancing the time by 1 hour, which

 \* is something we would have to do once a year for daylight saving time.

 \* The interesting case to consider is wrapping around to the next day, since

 \* the hour 23+1 would have to be represented as 0. Note the 24-hour clock.

 \*

 \* Precondition: The parameter t is a Time object with appropriate attribute

 \* values of hour, minute, second within their proper ranges.

 \* We assume that a copy constructor exists for the Time class.

 \* Postcondition: Return value is a Time object representing a time 1 hour

 \* after t.

 \*/

**3b** public static void incrementHour(Time t) {

 Time newTime = new Time(t);

 int hour = t.getHour();

 if (hour == 23)

 newTime.setHour(0);

 else

 newTime.setHour(hour + 1);

 return newTime;

 }

**4**

// Throughout this program, corrected errors have been identified.

// 1. Bad idea to have "import java.io.\*". Just import what you use.

// 2. Also, need to import classes from other packages besides io.

import java.io.FileNotFoundException;

import java.io.FileInputStream;

import java.util.Scanner;

// 3. The keyword class is not supposed to be capitalized.

// 4. It's good style to capitalize the name of a class.

public class CountWord {

 // 5. There is no need to have string constants for user prompts.

 // 6. The example code forgot the "[]" before args.

 // 7. A comment describing the entire program should be in javadoc,

 // which appears before and not after the main() header.

 /\*\* Purpose: To count the number of occurrences of a word in a

 \* text file. The user provides both the word and the text file.

 \*/

 public static void main(String [] args) {

 // 8. We should prompt the user for the file name before

 // grabbing the input.

 // 9. Console input comes from System.in, not System.out.

 // 10. It is better style to begin variable name with lowercase

 // letter.

 System.out.print("Enter the name of the text file: ");

 Scanner input = new Scanner(System.in);

 String fileName = input.nextLine();

 Scanner fileInput = null;

 // 11. It's good style to indent the body of a try block.

 // 12. If we can't open the file, program should halt, or

 // we could ask user to re-enter file name.

 // 13. The File class is not very helpful at reading text files.

 // Let's instead try FileInputStream.

 try {

 fileInput = new Scanner(new FileInputStream(fileName));

 } catch (FileNotFoundException e) {

 e.printStackTrace();

 System.exit(1);

 }

 // Use anything but a letter as a delimiter

 fileInput.useDelimiter("[^a-zA-Z]+");

 // 14. Again, we need to print prompt before grabbing input.

 System.out.print("Enter the word to be counted in the file: ");

 String word = input.next();

 // 15. We should also indent bodies of loops and if-statements.

 // 16. We should use meaningful identifiers,

 // e.g. instances instead of color

 // 17. We should be consistent about what identifier we use to

 // refer to the word we read from the input file.

 // 18. The count of words should be declared and initialized

 // before this loop.

 // 19. It's not necessary to print out all the words we

 // encounter in the file, except for debugging.

 int instances = 0;

 while (fileInput.hasNext()) {

 String fileWord = fileInput.next();

 // System.out.println(fileWord);

 if (word.equalsIgnoreCase(fileWord)) {

 instances++;

 }

 }

 fileInput.close();

 // 20. There should be space before the word 'appeared',

 // and a space after 'file' to make the output look good.

 // 21. It would be desirable to print 'time' in the singular

 // if the value of instances is 1.

 if (instances == 1)

 System.out.println("The word " + word + " appeared once in " +

 "the file " + fileName);

 else

 System.out.println("The word " + word + " appeared " + instances +

 " times in the file " + fileName);

 }

}

**5a** We need to design a Person class. A Person possesses a list of friends, who are other Person objects. A Person needs to have the ability (i.e. method) to extend a friend request to another Person, and this method needs the ability to accept or reject the friend request. Finally, a Person needs to be able to send a message to all friends.

**5b** At a minimum, the Person class will have three attributes: a name (String), a collection of friends (could be implemented as array of Person), and a buffer containing the message(s) from friends (a single String, or array of String).

The Person class should have a constructor. A Person begins with a name but no friends and a blank message buffer.

Instance methods:

First, we need sendRequest(Person p), which identifies whether Person p can be friends with me. If this method returns true, then we need to add p to the list of my friends, and make sure p adds me to the list of his/her friends. For simplicity, let’s assume that a friend request will always be honored, unless a person has reached some arbitrary maximum number. In this case, we need to have access to the number of friends p has.

Second, we need findNumberFriends( ) to help us implement sendRequest( ). This is an implementation detail inside sendRequest( ), so we don’t need to make this public.

We need an addFriend(Person p) method, so that the other person can add me to his/her list of friends.

We need sendMessage(String s), which broadcasts a message to all my friends. Each of my friends needs to append this message string into his/her buffer of messages.

Finally, we need some way of modifying the buffer of one of my friends, so we do this with a setBuffer(String s). This is an implementation detail inside sendMessage( ), so this does not need to be made public.

**5c** Person

 **-------------------------------------------**

- name: string

- friends: array of Friend

- buffer: string

-------------------------------------------

+ sendRequest(in p: Person) { query }

- findNumberFriends( ) { query }

+ addFriend(in p: Person)

+ sendMessage(in s: string)

- setBuffer(in s: string)

**6** Automobile

 **--------------------------------------**

- make: string

- model: string

- year: integer

- licencePlate: string

- mileage: integer

- gallons: double

- location: string

 **--------------------------------------**

 + getMake( ) { query }

 + getModel( ) { query }

 + getYear( ) { query }

 + getLicencePlate( ) { query }

 + getMileage( ) { query }

 + buyGas(in gallons: double)

 + drive(in gallons: double, in miles: integer)

 + getLocation( ) { query }

 + setLocation( ) { query }

**7**

Person

- name: string

- address: string

- ID: integer

+ getName( ) { query }

+ getAddress( ) { query }

+ getID( ) { query }

Course

- title: string

- code: string

- meetingTime: string

+ getTitle( ) { query }

+ getCode( ) { query }

+ getMeetingTime( )

 { query }

Faculty

- department: string

- salary: double

+ getDepartment ( )

 { query }

- getSalary( ) { query}

+ addCourse(in

 c:Course )

Student

- campusAddress: string

- major: string

+ getCampusAddress( )

 {query}

+ getMajor( ) { query }

+ addCourse( in c:Course)

+ dropCourse(in c:Course)

\*

1

\*

1

**8**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pass | q | rem | Verify rem >= 0 | Verify num = q \* den + rem |
| Initially | 0 | 17 | true | 17 = 0 \* 17 + 17, which is true |
| 1 | 1 | 13 | true | 17 = 1 \* 4 + 13, which is true |
| 2 | 2 | 9 | true | 17 = 2 \* 4 + 9, which is true |
| 3 | 3 | 5 | true | 17 = 3 \* 4 + 5, which is true |
| 4 | 4 | 1 | true | 17 = 4 \* 4 + 1, which is true |

**9a** Precondition: The array is nonempty, the elements are numerical, and each value in the array is in the appropriate range of values for the application (e.g. 0 to 100).

Postcondition: The method will return the average of the array elements.

**9b** Precondition: The height and weight are positive real numbers.

 Postcondition: The method will return a positive real number representing the corresponding BMI.

**9c** Precondition: The loan amount and interest rate will be positive real numbers. The number of months will be a positive integer. In particular, the interest rate will be expressed as a number of percent that loans are typically quoted at: for example the number 5.0 will mean the annual interest rate is 5%.

 Postcondition: The method will return the value of the loan payment.

**10** Yes, an assertion statement can help isolate bugs. For example, you can verify that the preconditions to a method are satisfied. If they are not, then you can conclude that a bug exists before the method was called.

Also, if the postcondition of a method guarantees some property of a return value (e.g. that it is positive), then this can be checked as well.

**11** This is an infinite loop. The loop will never terminate because the condition will always be true.

**12** Transaction

 **----------------------------**

 **-** date: string

 - time: string

 - amount: double

 - isChecking: boolean

 -----------------------------

 + getDate( ) { query }

 + getTime( ) { query }

 + getAmount( ) { query }

 + getType( ) { query }

**13** The value of numItems might be zero, negative, or larger than the size of the array. We should throw an exception in case the value of numItems is out of range.

 We should also take a step back and ask ourselves if we really need a method that averages the first numItems values of an array. If the only time we ever average an array is to average all of its elements, then this second parameter isn’t even necessary. Note that an array already has a built-in length attribute.

**14** a. The value of i is between 10 and 100, inclusive.

 b. The value of product is the product of these i factors: 1 \* 3 \* 15 \* … \* (2i – 1).

 c. The value of p is ac.

**15** The value of sum is the sum of all of the positive values among the first index values in array item.

**16** public class ex1\_8

{

 public static void main (String[] args)

 {

 int n = 5;

 int square = 1;

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

 {

 square = i \* i;

 System.out.println(square);

 }

 }

 }

**17** The bug is in the while loop:

 **a.)** Output is The floor of the square root of 64 is 7.

 **b.)** The while loop condition should be

 while(temp1 <= X)

 Debugging involves printing the values of temp1, temp2 and result at the top of the loop.

 **c.)** Supply user prompts and check the value of x to ensure that it is greater than 0.

**18** A loan application can be broken down into these steps:

 Get information about borrower

 Name

 Address

 SSN

 Residential history

 For each of the places where borrower has lived over the last several years:

 Get the start/end dates, address, name of landlord, amount of rent / mortgage info.

 Employment history

 For each of the jobs the borrower has had over the last several years:

 Get the start/end dates worked, job title and name of the boss.

 Balance sheet

 Ask borrower to enumerate assets and liabilities.

 Cash flow

 Ask borrower to indicate sources and amounts of income, and current debt payments.

 Get information about loan

 Down payment, if applicable

 Amount desired

 Duration of loan

 Interest rate

 Loan process

 Compute borrower’s net worth.

 Compute borrower’s net cash flow per month.

 What is the borrower’s credit score?

 Can the borrower afford the monthly payment?

 Does the borrower have a good credit history?

 Does the borrower have enough of a cushion to avoid missing a payment?

**19**

import java.util.Scanner;

public class Exercise0219 {

 public static void main(String [] args) {

 boolean needInput = true;

 Scanner kbd = new Scanner(System.in);

 int age = -1;

 while (needInput) {

 System.out.print("Please enter age: ");

 try {

 age = Integer.parseInt(kbd.nextLine());

 }

 catch(NumberFormatException e) {

 System.out.println("Error: age should be integer.");

 continue;

 }

 if (age < 0 || age > 122)

 System.out.println("Not a realistic value for age.");

 else

 needInput = false;

 }

 }

}