CSE 114 Fall 2020: Assignment 9

Due: December 2, 2020 at 11:59 PM

Read This Right Away

This problem set is due at **11:59 pm on December 2, 2020 KST**. Don't go by the due date that you see on Blackboard because it is in EST. Go by the one given in this handout.

Directions

- Download all the provided files from Blackboard.
- At the top of every file you submit, include the following information in a comment
    - Your first and last name
    - Your Stony Brook email address
- Please read carefully and follow the directions exactly for each problem.
- Your files, Java classes, and Java methods must be named as stated in the directions (including capitalization). Work that does not meet the specifications will not be graded.
- Your source code is expected to compile and run without errors. Source code that does not compile will not be graded.
    - You can get partial credit if your program is incomplete but does compile and run.
- You can create your program using a text editor (e.g. emacs, vim) or IntelliJ IDEA, whichever you prefer.
- Your programs should be formatted in a way that is readable. In other words, indent appropriately, use informative names for variables, etc. If you are uncertain about what a readable style is, see the examples from class and textbook as a starting point for a reasonable coding style.

What Java Features to Use

For this assignment, you are *not* allowed to use more advanced features in Java than what we have studied in class so far.

What to Submit

Combine all your .java files from the problems below into a single zip file following the instructions at the bottom of this page and upload this file to **Blackboard**.

Multiple submissions are allowed before the due date. Only the last submission will be graded.

Please do **not** submit `.class` files or any that I did not ask for.

Problem 1 (4 points)

Create files named: Kitchen.java, CookingAppliance.java, Blender.java, Toaster.java

Put each class described below in a separate file.

Create a simple class hierarchy consisting of an abstract class `CookingAppliance` and two concrete classes derived from `CookingAppliance` named `Blender` and `Toaster`. The `CookingAppliance` has two methods:

- a void method `use` that takes a variable number of arguments of type `Object…` (note the 3 dots indicating a variable number of arguments. You can work with these arguments as a standard array). So the method header should look like:
    - `public void use(Object... appliances)`
- a void method `clean` that takes no arguments

Implement the `use` and `clean` methods for the `Blender` and `Toaster` classes so that the sample code below, located in the main method of the `Kitchen` class,

```
CookingAppliance a = new Blender();
a.use("high");
a.clean();
a = new Toaster();
a.use("pizza", "low");
a.clean();
```

would produce the following output:

```
Running blender at high speed.
Cleaning the blender.
Toasting food on pizza setting at low temperature.
Cleaning the toaster.
```

Problem 2 (10 points)

Create files named NoiseMaker.java, Vehicle.java, Boat.java, Ship.java, Taxi.java, Pet.java, Cat.java, Fish.java, ObjectTester.java

Create and implement the following classes and interfaces:

- an interface `Noisemaker` that contains the void method `makeNoise()`
- an abstract class `Vehicle` that has an abstract void method `move()`, a data field `int price`, and which implements the `Comparable` interface. Create a constructor that initializes the `price` field via a parameter to the constructor (e.g., `public Vehicle(int price)`). Use the `price` field to determine whether one vehicle is “greater than”, “less than” or “equal to” another. Override the `public boolean equals(Object other)` method inherited from the `Object` class. Two vehicles are equal if their prices are equal.
- the concrete class `Boat`, which is a subclass of `Vehicle`
- the concrete class `Ship`, which is a subclass of `Boat` and which contains the concrete method `dropAnchor`. Override the inherited `move` method.
- the concrete class `Taxi`, which is a subclass of `Vehicle` and which implements `Noisemaker`

- the abstract class `Pet`, which has the data field `int age` and an abstract void method `eat()`. The `Pet` class implements the `Comparable` interface and compares two pet objects by age. Create a constructor that initializes the `age` field via a parameter to the constructor (e.g., `public Pet(int age)`).
- the concrete class `Cat`, which is a subclass of `Pet` and which implements `Noisemaker`
- the concrete class `Fish`, which is a subclass of `Pet`

Create a class `ObjectTester` with a main method that tests out your class/interface hierarchy. For the methods like `move`, `makeNoise`, `dropAnchor`, etc., provide simple print statements for the implementations. Below is part of a main method to get you started. You should add other objects and code to test your work. For example, you should write code that will NOT execute and then figure out why. The sample code below gives examples of this. Include such erroneous/invalid code in your file, but comment it out.

```
Pet p1 = new Cat(5);
p1.eat();

((Cat)p1).makeNoise();
Pet p2 = new Fish(4);
p2.eat();

Vehicle v = new Ship(1200);
v.move();
((Boat)v).move(); // notice which version of move is called
((Ship)v).dropAnchor();

Taxi t = new Taxi(1800);
t.makeNoise();
t.move();

Noisemaker n1 = new Cat(3);
n1.makeNoise();
Noisemaker n2 = new Taxi(500);
n2.makeNoise();

System.out.println(p1.compareTo(p2));
System.out.println(v.compareTo(t));
System.out.println(v.equals(t));

// Errors below. Think about why.
// System.out.println(t.compareTo(p1));
// Cat c = p1;
// Noisemaker n3 = p1;
```

Problem 3 (6 points)

Create file named: FractionCalculator.java

Write a program in a class named `FractionCalculator` that lets the user type in a simple math problem involving fractions, as shown in the examples below. The program evaluates the expression and prints the result. Your program must be able to handle addition, subtraction, multiplication and division of fractions. Use the provided `Rational` class to represent the fractions. You will need to use the `split` method in the `String` class to retrieve the numerator string and denominator string, and convert strings into integers using the `Integer.parseInt` method. Assume that the input is always formatted as below, as far as spacing is concerned. You do not need to check for invalid input.

Example #1:

```
Enter a math problem containing fractions: 3/4 + 1/5
3/4 + 1/5 = 19/20
```

Example #2:

```
Enter a math problem containing fractions: 3/4 - 1/5
3/4 - 1/5 = 11/20
```

Example #3:

```
Enter a math problem containing fractions: 3/4 * 1/5
3/4 * 1/5 = 3/20
```

Example #4:

```
Enter a math problem containing fractions: 3/4 / 1/5
3/4 / 1/5 = 15/4
```

Problem 4 (30 points)

All of the parts below require recursion to solve and should be placed in a `Recursion.java` file. Test your program with the provided `UseRecursion.java` file. Add additional test cases to ensure your methods are all working.

For all the following problems, they must be solved recursively, otherwise you will receive **0 points**. That means you need to have a recursive method call in your solution and should not use any loops. Using any loops will result in a 0 for that problem.

Part 1: Print an Integer in Reverse (4 points)

Write a recursive method `void reverseDisplay(int value)` that displays an `int` value reversely on the console. For example, `reverseDisplay(12345)` displays `54321`. Don't simply convert the `int` to a `String` - doing that would defeat the purpose of the exercise and you wouldn't really learn anything about recursion. You may not use any loops to write this method.

Part 2: Hailstone Sequence (4 points)

Consider the **hailstone sequence**: the integer sequence that results from manipulating a positive integer value n as follows:

If n is even, divide it by 2 (using integer division)
If n is odd, multiply it by 3 and then add 1

Repeat this process until you reach 1.

For example, starting with n = 5, we get the sequence 5, 16, 8, 4, 2, 1.

If n=6, we get the sequence 6, 3, 10, 5, 16, 8, 4, 2, 1.

If n=8, we get the sequence 8, 4, 2, 1.

Write the recursive method `int hailstone(int n)`, which computes and then returns the length of the hailstone sequence starting with n. You may not use any loops to write this method.

One possible recursive algorithm you can consider implementing is this:

```
if n = 1 then
  return 1
otherwise
  if n is even
    return 1 + the next hailstone value according to the "even" formula
  else
    return 1 + the next hailstone value according to the "odd" formula
```

Part 3: Count the Uppercase Letters (4 points)

Write a recursive method `int count(String str)` to return the number of uppercase letters in a `String`. You need to define the following two methods. The second one is a recursive helper method.

```
public static int count(String str)
public static int count(String str, int high)
```

For example, `count("Stony Brook")` returns `2`.You may not use loops in writing either method.

Part 4: Find a Multiple of 10 (4 points)

Write a recursive method `int findTimes10(int[] nums)` that analyzes an array of integers and returns the index of the first value whose successor is 10 times the original. If no such value exists, the method returns `-1`. You may not use any loops to write this method.

Examples:

`findTimes10(new int[]{1,5,3,5,50,2,4,6,60})` returns `3`
`findTimes10(new int[]{1,5,3,5,15,2,4,6,60})` returns `7`
`findTimes10(new int[]{1,5,3,5,15,2,4,6,5})` returns `-1`

Part 5: Replace Multiples of 5 (4 points)

Write a recursive method `void replaceMult5(int[] nums, int newVal)` that replaces all multiples of 5 in the `nums` array with the argument `newVal`. Make no other changes to the array. You may not use any loops to write this method.

Examples:

- `replaceMult5(new int[]{5,3,5,50,2,4,6,60}, 77)}`
  changes the argument to `{77, 3, 77, 77, 2, 4, 6, 77}`
- `replaceMult5(new int[]{5,3,-5,50,2,4,6,-60}, 99)}`
  changes the argument to `{99, 3, 99, 99, 2, 4, 6, 99}`
- `replaceMult5(new int[]{4,3,-8,33,2,4,6,-61}, 44)}`
  changes the argument to `{4, 3, -8, 33, 2, 4, 6, -61}`

Part 6: Mobius Ring (5 points)

Write a recursive method `String[] mobius(String s1, String s2)` that returns an array of `String` objects that represents all the possible Mobius ring readouts of the Mobius ring which has the words `s1` and `s2` written on the two sides. A Mobius ring is a surface with only one side formed by joining the ends of the strip together to form a loop where opposite sides of the page appear on the same side. Consider that on one side of the page we have the word "Hello" and on the other side we have "Java". The method would return an array containing the following strings, stored exactly in this order:

```
HelloJava
elloJavaH
lloJavaHe
loJavaHel
oJavaHell
JavaHello
avaHelloJ
vaHelloJa
aHelloJav
```

You may not use any loops to write this method.

Part 7: Teddy Bear Game (5 points)

Write a recursive method `int teddy(int initial, int goal, int increment)` that simulates a variant of the "Teddy Bears" game between the player (you) and your friend, and returns the minimum number of steps you would need to end up with goal bears. You start with an `initial` number of stuffed bears. During each step of the game you may perform one of the following actions:
(a) ask for and receive `increment` more bears from your friend;
(b) give away `increment` of your bears to your friend;
(c) if you have an even number of bears, you can give exactly half your bears to your friend; or
(d) if you have an even number of bears, you can take an additional number of bears equal to half your current number of bears.

If you fail to obtain the goal within 10 steps, the method returns `-1`. You may find it helpful to write helper methods to solve this problem. You may not use any loops to write any methods for this problem.

Examples:

`bears(10,4,2)` would return 2
`bears(9,5,3)` would return -1
`bears(15,4,2)` would return -1
`bears(20,3,5)` would return -1
`bears(40,5,6)` would return 3
`bears(30,56,5)` would return -1
`bears(30,55,4)` would return 3

Hints: There are four possible steps we can take with each move (two "giving" options and two "taking" options). The idea is to *recursively* explore trying each of those options and return the number of steps that each option required. (Keep in mind that some of those options are available only if we have an even number of bears.) If all four options return -1, we know that none of those paths was fruitful. Otherwise, we return the smallest value returned that is not -1. Keep in mind that we can make at most 10 moves, so this becomes a limit that will terminate the search (and the recursion).

Submission Instructions

Please follow this procedure for submission:

1. Place the deliverable source files into a folder by themselves. The folder's name should be CSE114_HW9_<yourname>_<yourid>. So if your name is Alice Kim and your id is 12345678, the folder should be named 'CSE114_HW9_AliceKim_12345678'

2. Compress the folder and submit the zip file.
    a. On Windows, do this by clicking the right-mouse button while hovering over the folder. Select 'Send to -> Compressed (zipped) folder'. The compressed folder will have the same name with a .zip extension. You will upload that file to the Blackboard.
    b. On Mac, move the mouse over the folder then right-click (or for single button mouse, use Control-click) and select **Compress**. There should now be a file with the same name and a .zip extension. You will upload that file to the Blackboard.
3. Navigate to the course Blackboard site. Click **Assignments** in the left column menu. Click **Assignment 9** in the content area. Under **ASSIGNMENT SUBMISSION**, click **Browse My Computer** next to **Attach Files.** Find the zip file and click it to upload it to the web site.