# CCPS 109 Weekly Labs

This document contains the specifications for the labs of the course **CCPS 109 Computer Science I**, as taught by Ilkka Kokkarinen.

## General instructions for all labs

Each lab consists of four problems, and **solving each problem correctly so that it compiles, works and passes my automated JUnit tests (the last Swing lab will be manually graded by the instructor with the eyeball method) is worth half point**. This gives you a maximum total of 20 points for the labs. Email each lab to the instructor to *ilkka.kokkarinen@gmail.com* as the complete BlueJ project archive. (This allows the instructor to grade your submission in literally ten seconds flat with couple of clicks of the mouse. **There are over 70 of you enrolled already, so it is absolutely essential that you submit your labs in the required form!**)

In all problems, you may assume that the **arguments passed to each method are legal as defined in the problem specification**. **Your methods don't need to be able to handle or recover from any illegal arguments.** Of course, in real programming, it is good to make your methods **robust** so that they don't crash when given illegal inputs, but handle them gracefully. However, proper error handling using exceptions will be taught in CCPS 209, so I cannot require you to do that in this course.

Once you have written each method, give it a small "sanity check" by interactively creating an object in BlueJ and calling that method with some arguments that are simple enough so that you can easily see what the correct result should be, but also not too simple, so that the method actually has to do some non-trivial work to produce the correct answer. If the method returns an unexpected answer, try to reason what is wrong with the logic of the method.

Except for the Swing programming lab 10, **the instructor provides a JUnit test suite that you must use to verify that your code works correctly**. Include that tester class in your BlueJ project alongside the class that the lab specification asks you to write. Once you have written all of the required methods in your class, you can compile the test class. (**The tester classes do not have syntax errors: if the tester does not compile, it means that some method in your class is missing or misspelled, or that you have copypasted the tester class incompletely.**) BlueJ displays JUnit test classes in green instead of the usual yellow. The popup menu of the test class allows you to run either all tests at once, or only one individual test. Each method that passes my JUnit tester gets 0.5 marks for that question. **There are no partial marks given for any methods that do not pass the tester.**

**You are not allowed to modify the provided JUnit test class in any way**. Especially any modification that intentionally attempts to make your non-working methods seem like they pass my tests is considered blatant **academic dishonesty** and will automatically, with a zero tolerance policy, be punished by **forfeiture of all lab marks from this course**.

Of course, you would usually like to test some of your methods before writing all of them. Instead of modifying the tester, you can always implement every method initially as an empty **stub** where each method does nothing but just returns zero or some other default answer suitable for the method's return type. Of course this stub has a zero chance of passing the tester, but the point is that **having the required method stubs in your class allows the test class to compile**, so that you can use it to test your methods one by one as your write them, instead of first writing all methods to completion and then trying to test and debug them all at once. When you get one method to work, move on to put some "meat" in the method body of the next stub.

One more very important thing: **Silence is golden!** In every lab, make sure that **your methods are quiet** so that **they don't do any `System.out.println` console output when submitted**, but just quietly `return` the asked result. During the development, you can use console output for some "quick-and-dirty" debugging (although in many cases, it would be much better to use BlueJ **breakpoints** and stepping through the code with debugger for this purpose), but once you get your method working, **take out all those console output statements, or at least comment them out.** Otherwise, running the JUnit test suite might take a very long time to complete, since textual output to the graphical screen will certainly become very sluggish if millions of lines get printed.

## Lab 1

In the first lab, we practice creating classes and objects and representing data and their operations in them, and writing methods that work on data given to them, similarly to the example class Length.

Write a class *Rectangle* that has two *int* fields *width* and *height*, and a public constructor *Rectangle(int w, int h)* that assigns its arguments to these fields. Then, write the public accessor ("getter") methods *int getWidth()* and *int getHeight()* to return the values of these two fields, and public accessor methods *int getArea()* and *int getPerimeter()* that compute and return the area and the perimeter length of the rectangle. Last, write a mutator method *void flip()* that internally swaps the values of the *width* and the *height* of the rectangle with each other. (For example, flipping a 2*5 rectangle would turn it into 5*2 rectangle.)

Try out your class in BlueJ by creating an object from it. Play around with your object by calling its methods and see if they return the correct answers. When you are satisfied with this, copy the JUnit tester class TestRectangle.java into the same project, compile it (since it is a JUnit tester, the compiled class will show up as a green box instead of yellow) and run the tests from the right click popup menu. If some test does not give you the green check mark, debug your code. (For debugging purposes, you might also want to implement a *toString* method.)

After this question (worth full two points this first time, just to get you started), after you got a better idea how classes and objects work, register to CodingBat and start doing some warm-up problems to see how CodingBat works, and more importantly, to practice writing logic and conditions in your methods. These

problems are not graded in the lab, and the warm-up problems will not become part of the 50 problems that you have to solve in CodingBat to get to enter the final exam. **Note that in CodingBat, you always write only the one method whose signature and braces are already given to you. CodingBat then writes the rest of the class around your method for its internal compilation and testing purposes.**

## Lab 2

In this lab, we practice writing if-else conditions with a couple of methods inspired by familiar card games. An individual playing card is represented as a string of two characters so that the first character is the rank (from "`23456789TJQKA`", and note that ten is encoded as letter `T` to make all card ranks single letters) and the second character is the suit (from "`cdhs`"). For example, "`Jd`" would be the jack of diamonds, and "`4s`" would be the four of spades. A hand made up of multiple cards is given as string encoding all those cards consecutively, for example "`Kh3h7s8h2h`" for a five-card hand that happens to have one spade and four hearts. **Note that the cards can be listed inside the string in any order, not necessarily sorted by suit or rank.** The suits and ranks are also **case sensitive**, with the rank always given as a digit or an uppercase letter, and the suit always given as a lowercase letter from the four possible letters *cdhs*.

**Important: when calling a method that takes a String argument interactively in BlueJ, you need to enter the argument string in double quotes in the BlueJ dialog! This is because the BlueJ method call dialog box expects you to give it a Java expression that it evaluates and passes the result to the method as argument.**

Write the following methods in the class named *CardProblems*, and then use the JUnit test class TestCardProblems.java to test them. Remember that the *String* class has a handy method *charAt(int idx)* that returns the character in the position *idx* of that string.

1. The card ranks inside the hand are given as characters inside the string, but sometimes it would be more convenient to handle these ranks as numerical values from 2 to 14. The **spot cards** from two to ten have their numerical values, and the **face cards** jack, queen, king and ace have the values of 11, 12, 13 and 14, respectively. Write a method *int getRank(char c)* that returns the numerical value of the card given as the character parameter *c*. For example, when called with argument `'Q'`, this method would return 12. **You must write this method as an if-else ladder.**
2. The complete hand classifier for **poker** would be far too complicated for us to write in this course, so let's just look at a couple of instructive special cases. First, write a method *boolean hasFlush(String hand)* that checks whether the five-card poker hand given as 10-character string is a **flush**, that is, all five cards are of the same suit. Ranks don't matter for this problem, since for simplicity, we don't care about the rare possibility of straight flushes. (Hint: The suits are given in the positions 1, 3, 5, 7 and 9 of the parameter string.)
3. A bit trickier question is to determine whether the given five-card poker hand contains **four of a kind**, that is, four cards of the same rank. Write a method *boolean hasFourOfAKind(String hand)* to determine this. (Hint: This time the suits don't matter, and the ranks are given in positions 0, 2,

4, 6 and 8 of the parameter string. There are many possible ways to organize the logic of this method.)

4. In the exotic draw poker variation of **badugi**, the players try to draw a four-card hand in which no two cards have the same suit or rank. Write a method *boolean hasFourCardBadugi(String hand)* that checks whether the given four-card hand is a badugi, that is, **all four ranks are different, and all four suits are different**. For example, given the hand "2d7cKhJs" the method would return *true*, but given the hands "4d4cKhJs" (that has two fours) or "2d7cKhJd" (that has two diamonds), the method would return *false*.

## Lab 3

We continue with problems inspired by various classic and popular card games, this time demonstrating the use of loops and repetition to solve problems. Create a class named *MoreCardProblems* and write the following methods there. Again, you must use the JUnit test class TestMoreCardProblems.java to test your methods.

1. In the game of **contract bridge**, each player is dealt a hand with thirteen cards. The basic "Milton Work point count" for hand evaluation, about the first thing that any bridge beginner learns, estimates the strength of the hand for the bidding purposes the following way: each ace is worth 4 points, each king is worth 3 points, each queen is worth 2 points, and each jack is worth 1 point. Write a method *public int bridgePointCount(String hand)* that, given a bridge hand as a string of 26 characters, computes and returns this point count.
2. In the game of **gin rummy**, once the hand is over after **knocking and melding**, each player scores "deadwood" penalty points for the cards that he could not get rid of. Same as in blackjack, aces count for one point, all face cards count for ten points, and the spot cards count according to their face value. Write a method *public int countDeadwood(String hand)* that computes the deadwood points in the given hand. The hand can have *any* number of cards up to the full 52, so **you must use a for-loop** to go through the individual cards. (Note that "any number" here includes zero! Your method should correctly return zero deadwood points for an empty hand.)
3. Back to contract bridge. The shape of the hand (that is, how "balanced" or "unbalanced" it is) is often just as important as its raw point count. Write a method *public String bridgeHandShape(String hand)* that creates and returns a four-element string that tells how many spades, hearts, diamonds and clubs, **in this order**, there are in the hand. This result string should contain these four counts in order separated by commas, with exactly one space after each comma. Note that there should be no comma or space after the number for clubs. For example, when called with the argument "`Kh6c2cJdJcAd7h3d8dQdTsTh3h`", the method would create and return the string "`1, 4, 5, 3`" since this hand contains one spade, four hearts, five diamonds and three clubs.
4. In poker, a **full house** consists of some three cards of equal rank, and some pair of two equal cards of another rank: for example, three tens and two fours, or three queens and two kings. Write a method *public boolean hasFullHouse(String hand)* that checks whether the given five-card poker hand is a full house. (Again, remember that the cards can be listed in the hand in any order.

This problem is a bit difficult, which is why it is the last one today. As always, there are many possible working ways to organize the decision making logic of this method.)

## Lab 4

In this lab, we shall take a closer look at the *String* class and its operations, and use it to practice writing logic with loops. Create a class *StringProblems* and write the following methods in it. Again, you must use the JUnit test class TestStringProblems.java to test your code. To serve as test data needed by the JUnit tester class, save the file warandpeace.txt into your project folder **exactly as given** (**do not open this file with some text editor and save it from there formatted in a different way**). (Dropbox link) In all the methods that ask you to create a long result string by repeatedly adding individual characters to the result, for efficiency reasons you should use the mutable *StringBuilder* optimized for this purpose, and then convert the final result to a *String*. If you just use the bare *String* to which you add the characters one by one, your method will be inherently inefficient and the JUnit test methods will take a very long time to run.

1. *String removeDuplicates(String s)* that creates and returns a new string with all the consecutive occurrences of the same character turned into a single character. For example, when called with the string "aabbbbbabbbbccccddd", this method would create and return the string "ababcd". (Hint: loop through the characters of the string, and add the character to the result only if it is different from the previous character. Be careful at the beginning of the string.)
2. *int countWords(String s)* that counts how many words there are in the parameter string *s*. Each word starts at a non-whitespace character that either begins the entire string, or is immediately after some whitespace character. (Use the static utility method isWhitespace in the utility class Character to identify the whitespace characters. There are way more possible whitespace characters than just the common one that you get by pressing the space bar, and especially non-whitespace characters there are tens of thousands of, and in today's interconnected world you can't simply assume that only the 26 letters of English and the basic punctuation characters exist!) For example, the string `"  Həllo world! H☺w      are yôu today?"` contains six words, the boldface here indicating which characters begin a new word.
3. *String convertToTitleCase(String s)* that creates and returns a new string where the first letter of each word is converted to title case, using the same definition of a "word" as in the previous problem. All other characters are kept as they were. (Use the static utility method toTitleCase of Character to do this character conversion.) For example, when called with the string "The quick brown fox jumped ovɛr a lazy dog!", this method would return the string "The Quick Brown Fox Jumped Ovɛr A Lazy Dog!"
4. *String stretch(String s, int n)* that creates and returns a new string so that its first and the last characters are same as in the parameter string *s*, but each individual character in between is duplicated *n* times. For example, when called with arguments "world" and 3, this method would create and return the string "wooorrrllld". (Make sure that your method works correctly also for any string of zero or one characters, and in the edge case where $n == 0$. For example, when called

with the arguments “world” and 0, this method should return the string “wd”. Given arguments “xy” and 1000000000, this method should immediately return “xy” instead of crashing due to running out of memory.)

## Lab 5

This week, it is time to practice arrays so that our methods can process millions of items of data, giving our loops and conditions something useful to do. Create a class *ArrayProblems* and write the following array methods in it. Again, you must use the JUnit test class TestArrayProblems.java to test out your methods.

1. *int[] everyOther(int[] a)* that creates and returns an array that contains the elements from the **even-numbered positions** of the array *a*, that is, the elements *a[0], a[2], a[4],...* Make sure to compute the length of the result array exactly right so that there is no extra element in the end. (You need to consider the possible odd and even lengths of *a* separately. The integer arithmetic remainder operator **%** might come handy here.)
2. *int countInversions(int[] a)* that counts how many **inversions** there exist inside the array *a*. An inversion is a pair of indices (*i*, *j*) to the array so that $i < j$ and $a[i] > a[j]$. Use two nested *for*-loops to loop through all possible pairs of array indices. (In the analysis of sorting algorithms, the inversion count is an important measure of how “out of order” the elements of an array are compared to a fully sorted array that has zero inversions.)
3. *void squeezeLeft(int[] a)* that **modifies the contents of its parameter array** *a* by moving all its nonzero elements to the left as far as it can, writing over the zero elements that precede them. For example, if called with the parameter array {0, 1, 0, 0, 3, -2, 0, 7}, the contents of this array after the call would be {1, 3, -2, 7, 0, 0, 0, 0}. Note that **this method does not return anything, but it modifies the contents of the array object given to it as argument**. (For an extra challenge, you can try make your method work **in place** so that it doesn’t internally create another array to use as temporary workspace. For an even harder challenge for the hardcore algorists, make your method work in **one pass** through the array.)
4. *int[] runDecode(int[] a)* that takes an array that consists of a series of element values followed by the lengths of the run of those elements, listed in alternating locations. This method creates and returns a new array that contains these elements, each element repeated the number of times given by its run length. For example, when called with the array {3, 8, 1, -7, 0, 42, 4, -3} (read this out loud as “three eights, one minus seven, zero forty-twos, and four minus threes”), your method would create and return the array {8, 8, 8, -7, -3, -3, -3, -3}. Again, make sure that the array returned by your method has the exact right length.

   (You can assume that the length of the parameter array is even. A run length can never be negative, although it can be zero. As a possibly tricky edge case, the result array could even end up being empty, if every run has zero length!)

## Lab 6

We continue writing code for arrays. First, let's define some useful terminology to use with this week's problems. Based on how the value in some array position *i* compares to the value in its **successor** position *i* + 1, each position *i* in the array *a* is classified as an **upstep** if *a*[*i*] < *a*[*i* + 1], a **downstep** if *a*[*i*] > a[*i* + 1], and a **plateau** if *a*[*i*] == *a*[*i* + 1]. The last position of the array has no such classification, since there is no successor element that we could compare its value with.

Armed with these definitions, create a class *ArrayShapeProblems* and write the following methods in it, after which you should again use the JUnit test class TestArrayShapeProblems.java to try them out.

1. *int countUpsteps(int[] a)* that counts how many positions in the parameter array *a* are upsteps, and returns that count.
2. *boolean sameStepShape(int[] a, int[] b)* that checks whether the two parameter arrays *a* and *b* are the same overall shape in that every position *i* has the same classification (upstep, downstep or plateau) in both arrays. (Your method may assume that *a* and *b* have same `length`.)
3. *boolean isSawtooth(int[] a)* that checks whether the parameter array *a* has a **sawtooth shape**, meaning that it has no plateaus, and that each upstep is immediately followed by a downstep and vice versa. (Hint: it is much easier to determine whether the given array is **not** sawtooth, and then negate this answer.)
4. *boolean isMountain(int[] a)* that checks whether the parameter array *a* is **a mountain**, meaning that it contains no plateaus, and that it initially contains zero or more upsteps, which are then followed by zero or more downsteps. Note that by this definition, an array that contains nothing but upsteps is a mountain, as is an array that contains only downsteps. (Hint: this method should **operate in a single pass**. Don't try to make this too complicated: simply think of the sentence "Once you have gone down, you may not go up again" and build your logic on that.)

## Lab 7

Here are some more problems about writing loops and using strings. Create a class *MoreStringProblems* and implement the following methods in it. Once again, the JUnit class TestMoreStringProblems.java must be used to test your code.

1. *String uniqueCharacters(String text)* that creates and returns a *String* that contains each character that occurs in the given *text*, so that each character is included in this result only once. These characters must occur in the result in the exact same order as they first occur in the original string. For example, when called with argument `"Hello there, world!"`, this method would return `"Helo thr,wd!"` (Again for efficiency, you should use a *StringBuilder* inside the method to build up the result.)
2. *int countOccurrences(String text, String pattern)* that counts how many times the *pattern* occurs

inside the *text*. Remember to properly count also the **overlapping** occurrences of the pattern: for example, the string `"bababababa"` contains three occurrences of the pattern `"baba"`. You can assume that the *pattern* is not empty, although the *text* can be, in which case, your method needs to properly return the correct answer 0 instead of crashing. (Hint: testing if pattern is longer than text and returning zero in that case solves that and many other tsituations.)

3. *String cyclicLeftShift(String s, int k)* that creates and returns a string in which character of *s* has been moved left *k* steps, so that the characters that would "fall off" the left end are appended to the right. For example, when called with the parameters `("Hello world", 3)`, the method would return `"lo worldHel"`. Design your method so that it works quickly even if *k* is greater than the length of *s*, maybe even as large as +1,000,000,000. (Hint: rotating any string by its length effectively does nothing, and therefore rotating any string by any integer multiple of its length will also do nothing. Therefore all you need to do is… what?)
4. *int parseInt(String s)* that extracts the integer (either positive or negative) whose digits are given as characters in the parameter string *s*. This method should therefore work exactly like the standard library method *Integer.parseInt*. Of course, so that this problem wouldn't be trivial to begin with, **you are not allowed to call that existing method** (or the equivalent *String.valueOf* method) from your code, but must implement the internal logic all by yourself using only the basic *String* and integer operations of the Java language. For simplicity, you can assume that the parameter string *s* consists of one or more digit characters, possibly preceded by a minus sign, so that you don't need to handle underscore separators, legal in Java integer literals since Java 7. As always, you can also assume that input is always legal, so you don't need to detect non-digit characters or handle the integer overflow.

## Lab 8

This week, it is time to wrangle two-dimensional arrays. Write the class *TwoDeeArrayProblems* with the following methods to be tested with the class TestTwoDeeArrayProblems.java. As usual, all these methods must be silent so that they output nothing on the console, but for your own debugging purposes during the development, note that an easy way to print out the contents of the two-dimensional array `arr` would be `System.out.println(java.util.Arrays.deepToString(arr));`

1. *double[][] transpose(double[][] a)* that creates and returns a two-dimensional array *b* that has as many rows as *a* has columns, and as many columns as *a* has rows. (For this problem to be meaningful and possible, it is guaranteed that each row of *a* has the same number of elements.) Each element *b[i][j]* of the result should equal the value of the original element *a[j][i]* in the mirror image position. For example, if the original array contains 2 rows and 3 columns, with the elements

```
1     4     -2
-5    0     3
```

the array returned by this method would contain 3 rows and 2 columns, with the elements

```
1     -5
4     0
-2    3
```

so in the array transpose operation, the rows become columns, and columns become rows.

2. *double[] minValues(double[][] a)* that creates and returns the one-dimensional array *b* whose length equals the number of rows in *a*. Each element *b[i]* should equal the smallest element in the *i*:th row of the parameter array *a*, or if that row is empty, equal 0.0.
3. *int[][] zigzag(int rows, int cols, int start)* that creates and returns a two-dimensional array of integers with the given number of *rows* and *cols*. This array should contain numbers ascending from *start* listed in order in the consecutive rows, except that every row whose row index is odd is filled in reverse order. For example, if *rows* is 3, *cols* is 5 and *start* is 14, this method would create and return a two-dimensional array with the elements

```
14    15    16    17    18
23    22    21    20    19
24    25    26    27    28
```

4. *double maximumDistance(double[][] pts)* that is given a two-dimensional array *pts* that is guaranteed to have at least two rows, and each row is guaranteed to have exactly three elements. Each row of this array *pts* gives the (*x*, *y*, *z*) spatial coordinates of a point in the three-dimensional space. This method should find the two points that have the maximum distance from each other, and return this distance.

   To calculate the distance between two points `p1` and `p2` given as `double[]`, use the method

```
private double distance(double[] p1, double[] p2) {
  double sum = 0.0;
  for(int i = 0; i < p1.length; i++) {
    sum += (p1[i] - p2[i]) * (p1[i] - p2[i]);
  }
  return Math.sqrt(sum);
}
```

## Lab 9

In this penultimate lab, you get to try your hand with some recursion operating on arrays and subarrays. All of the following methods of the class *RecursionProblems* must be fully recursive, with **no loops allowed at all!** (If-else statements are allowed, as they are not loops.) Because these methods are quite short, this last time there are six methods to write instead of the usual four. (The scoring rule then changes

that you start gaining any points for this lab at the *third* method that passes the tester.)

As in general with recursion, you should write these methods so that you don't use any additional fields in the class to store the intermediate results, but you pass all the data that your methods need as parameters going down, and return them as results coming up. (Using local variables inside the method is perfectly acceptable.) Again, the JUnit test class TestRecursionProblems.java must be used to test the correctness of your methods.

1. *boolean allEqual(int[] arr, int start, int end)* that checks if all elements in the subarray of *arr* from *start* to *end*, inclusive, are equal. Note that for the base case, all elements of an empty (that is, where *start* > *end*) or one-element subarray are equal by definition. After all, a subarray must by definition have at least two elements for it to have two different elements! (This principle generalizes to the counterintuitive but nonetheless true observation that any universal claim that you make about the members of an empty array is trivially true by definition.)
2. *void arraycopy(double[] src, int start, double[] tgt, int start2, int len)* that copies *len* elements from the location *start* of array *src* to the location *start2* of array *tgt*. In other words, this method works exactly the same way as the method *System.arraycopy*. (But of course you are not allowed to call this existing method.) Hint: when *len* is less than one, do nothing. Otherwise copy the first element, and then recursively copy the remaining *len* - 1 elements.
3. *boolean linearSearch(int[] arr, int x, int start, int end)* that checks if the unsorted array *arr* contains the element *x* anywhere in the subarray from *start* and *end*, inclusive.
4. *void reverse(int[] arr, int start, int end)* that reverses the order of the elements in the subarray of *arr* from index *start* to *end*, inclusive. Note that this method does not create and return a new array, but modifies the contents of the parameter array *arr* in place by reassigning its elements.
5. *public void parityPartition(int[] arr, int start, int end)* that modifies the subarray of *arr* from *start* to *end*, inclusive, in place so that all odd numbers are first together in one bunch, followed by all even numbers in another bunch. The odd elements can end up in any order you want inside the first bunch, as do the even elements. For example, if your array *a* is currently [2, -1, 3, 4, 8, 5, -7], after the call *parityPartition(a, 0, 6)*, the contents of this array might become [3, -1, -7, 5, 8, 4, 2]. (Hint: after testing for the base case of subarray that has only zero or one elements, the parities of the first and the last element of the subarray, whichever way they might be, will somehow allow you to continue the recursion with a smaller subarray.)
6. *int countRuns(int[] arr, int start, int end)* that counts how many **runs** (consecutive blocks of equal elements) there are in the subarray of *arr* from index *start* to index *end*, inclusive. For example, the array [4, 4, 4, -2, 0, -8, -8, 3, 3, 3] has five runs, whereas the arrays [5, 5, 5, 5] and [42] each have only one run. An empty subarray (that is, when *start* > *end*) has zero runs. (Hint: count how many elements are different from the next element.)

## Lab 10

Let us finish off this course with flamboyant flourish by writing some interactive GUI components using

Swing, this task also serving as a segway to CCPS 209 by using **inheritance** to create powerful GUI component classes with just a couple of lines of code. In this lab the questions two, three and four that ask you to write an event listener, you **must use some kind of nested class to do the listening**, and should not make the component class itself to be the listener. Yes, I know that many example programs found online do it that way, but it is still so very wrong, wrong, wrong! Any solution where the component class itself implements the listener will be summarily rejected on the spot.

Unlike in the previous labs, **in this lab each one of the four questions should be a separate class of its own, created as a separate class file inside the same BlueJ project. There is no need to place each class in a separate project here.** Also, there is no JUnit tester provided, but Swing components are tested visually.

1. Let's start out easy so that there is no user interaction yet. Write a Swing component *MyLabels* that extends *JPanel* and contains three *JLabel* instances that display "One", "Two" and "Three", respectively. Your component should prefer to have a size of 200*100 pixels, and have an etched border around it. Write a *main* method that creates and opens a *JFrame* instance that uses *FlowLayout* and contains three separate instances of *MyLabels*. (Inside the frame, there will therefore be a total of nine labels showing; that is, three labels inside each of your three *MyLabels* components.)
2. Next, a component similar to the first one except with user interaction and event handling. Write a Swing component *MyButtons* that extends *JPanel* and contains three *JButton* instances titled "Red", "Green" and "Blue". Your component should prefer to be 250*100 pixels in size, and use a nested ActionListener subclass to listen to the action events of these three buttons. Pressing a button should change the background colour of the entire *MyButtons* component to that colour. (Use the method *setBackground*.) Write a *main* method for your class that creates and opens one *JFrame* instance that contains a total of six separate *MyButtons* components, arranged in a GridLayout to sit in two rows and three columns.
3. Write a Swing component *Head* that extends *JPanel* whose method *paintComponent* draws a simple human head. This doesn't need to look fancy so you can just use basic ellipses and rectangles, but of course those who are interested can try out some niftier shapes in java.awt.geom. This component should have one JCheckBox instance labelled "Hat". If this checkbox is selected, the head is drawn with a hat on, otherwise the bare head is drawn without a hat. Write an inner ItemListener class to listen to the item events of the checkbox so that whenever the user changes the state of the checkbox, the component is *repaint*ed. In the *paintComponent* method, draw the hat if the checkbox is currently selected, and otherwise, well... just don't draw the hat. Simple as that. (Note that **you should never do any drawing in the listener methods**. Just call *repaint* and let the *paintComponent* method draw the component as it looks like after the change.)
4. Write a Swing component *Adder* that contains two JTextField instances, one JButton labelled "Add" and one JLabel that initially contains empty spaces. When the user presses the button, the component should read the numbers from the two text fields, add them up and show the result in the label. Note that your component doesn't need to listen to the action events of the text fields, but only those of its button. For simplicity, you may assume that the human user will always enter a proper number in both textfields, so there is no need to catch and handle malformed input. (That

one we shall again leave for CCPS 209.)