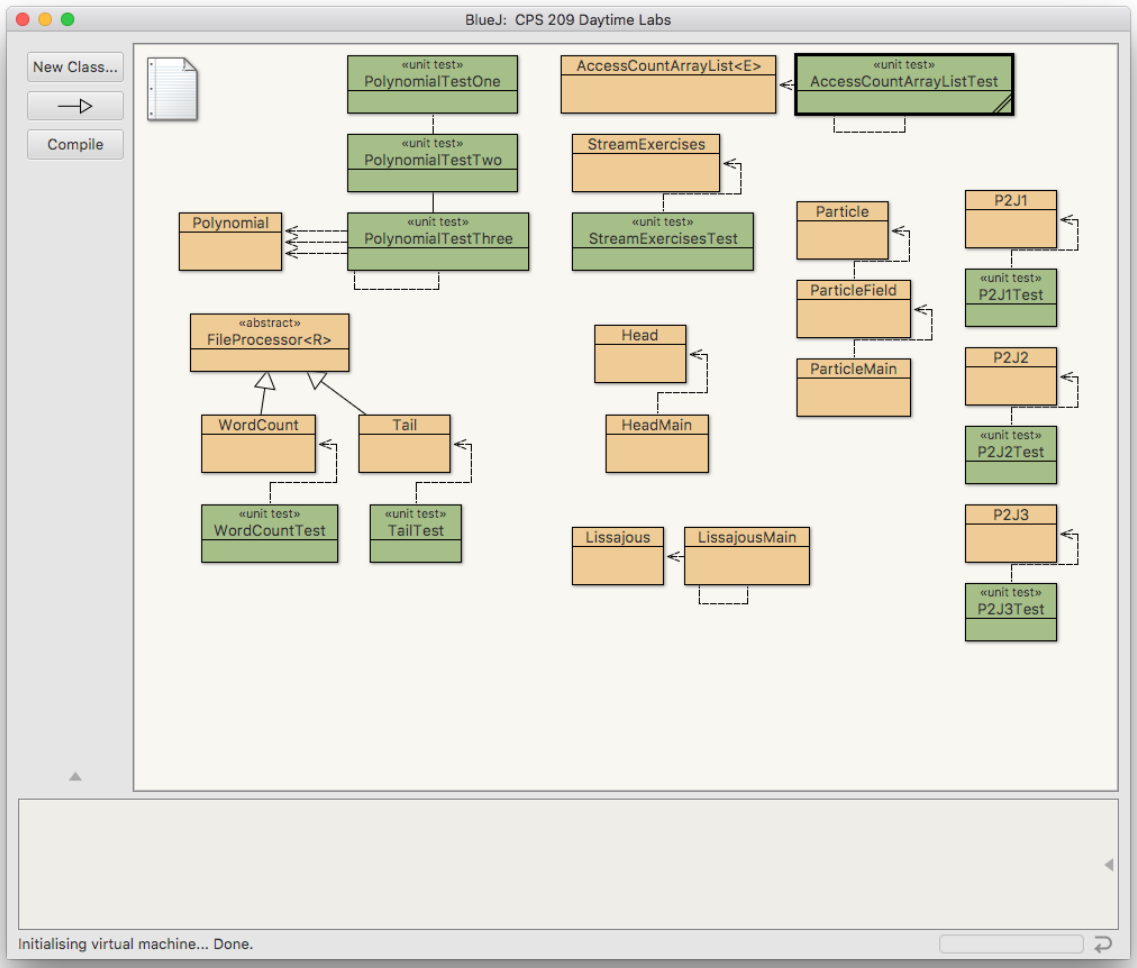
CCPS 209 Labs

This document contains the graded labs for the course **CCPS 209 Computer Science II,** as taught by Ilkka Kokkarinen. This document contains a total of thirteen labs. Out of these thirteen, it is sufficient to do **any ten labs of your choice** to earn the maximum lab mark of 20 points for this course. However, **the instructor strongly recommends that you complete all three transition labs 0(A), 0(B) and 0(C)** to become proficient in Java so that you learn to think and type in that language without having to use conscious thought on the language itself. This will allow you to see the forest instead of just the individual trees and concentrate on the actual important topics in the remaining ten labs, instead of getting mired in the nitty gritty language details.

You must create one BlueJ project folder named "209 Labs" inside which you write all of these labs. That's right, **all your labs go in one and the same project folder.** In this same BlueJ project, you should also add the JUnit test classes provided by the instructor. Then, using Finder (if working on Mac) or Windows Explorer (if on Windows), you should copy the text file `warandpeace.txt` inside your project folder. You should download and save this text file directly into your project folder from the DropBox link without going through any kind of text editor or copy-pasting. **Do not edit this text file in any way, not even to add or remove some little whitespace character**.

The BlueJ project that you work on, once it is ready for submission, should look something like the following screenshot taken from the private model solution of the instructor, except that you need to submit only ten labs of your choice instead of all thirteen, and of course the boxes representing the individual classes can be arranged any which way inside your project window.



Each of these labs is worth the same two points of your total course grade. For the labs 5, 7 and 9 that ask you to write a Swing component, these two marks are given for these components passing the visual inspection of the behaviour specified in the lab. The other labs must cleanly pass the entire JUnit tester to receive the two points for that lab. These two points are **all or nothing** for that lab, and **no partial marks whatsoever are given for labs that do not not pass these tests.** Since you have a total of thirteen labs to choose your ten labs from, choose the labs that you are able to get to work.

To compensate for this strict policy of correctness, **all the labs have the same deadline of noon after the day of final exam**. Before that, there is no rush to complete any lab, so you can take your time to get each lab to work smoothly so that it will pass the JUnit tests. You can then simply come to each lab session to work on and get help for whichever lab you are currently working on, to make the most efficient use of our time.

In all these labs, silence is golden. Since many of these JUnit testers will test your code with a large number of pseudo-randomly generated test cases, your methods should be absolutely silent and print nothing on the console during their execution. **Any lab that prints anything at all on the console during the execution will be unconditionally rejected and receive a zero mark.**

You may not modify the JUnit testers provided by the instructor in any way whatsoever, but your classes must pass these tests exactly the way that your instructor wrote them. Modifying a JUnit tester to make it look like that your class passes that test, even though it really does not do so, is considered **serious academic dishonesty** and will be automatically penalized by the forfeiture of all lab marks in the course. These labs have now been used for one semester and several students, working independently of each other, have successfully implemented their code to pass all these tests.

Once you have completed all the labs that you think you are going to complete before deadline, you will **submit all your completed labs in one swoop** as the entire "209 Labs" BlueJ project folder that contains all the source files and the provided JUnit testers, compressed into a zip file that you upload into the assignment tab on D2L. Even if you write your labs working on Eclipse or some other IDE, it is compulsory to submit these labs as a single BlueJ project folder. **No other form of submission is acceptable**.

ACCIDENTS HAPPEN, SO MAKE FREQUENT BACKUPS OF THE LAB WORK THAT YOU HAVE COMPLETED SO FAR. SERIOUSLY. THIS CANNOT BE EMPHASIZED ENOUGH NOT ONLY IN THIS COURSE, BUT IN ALL OUR LIVES.

Lab 0(A): Arrays and Arithmetic

JUnit: P2J1Test.java

These three new transition labs 0(A), 0(B) and 0(C) are designed to help you translate your existing Python knowledge into equivalent Java knowledge in both your brain and fingertips. You may already be familiar with these problems from the instructor's CCPS 109 course, both the new Python version and the old Java version, but even if you are not, these problems are self-contained and can be solved by anybody aspiring to learn Java from either Java or some other language background. Each transition lab consists of four `static` methods for you to write, there is no object oriented thinking involved yet. All four methods must pass the entire JUnit test to receive the two points for that lab. There are no partial marks given.

Inside your fresh new BlueJ project, create the first class that **must be named** exactly `P2J1`. Erase the nonsense template that BlueJ fills inside the class body in its misguided effort to "help" you, and in its place, write the following four methods.

```
public static int fallingPower(int n, int k)
```

Python has the integer exponentiation operator ** conveniently built in the language, whereas Java unfortunately does not have that one. (In both languages, the caret character ^ means a totally different operator of **bitwise exclusive or** that has nothing to do with exponentiation.)

However, in the related operation of **falling power** that is useful in many combinatorial formulas and denoted syntactically in mathematical notation by underlining the exponent, each term that gets multiplied into the product is always one less than the previous term. For example, the falling power $8^{\underline{3}}$ would be computed as the product 8 * 7 * 6 = 336. Similarly, the falling power $10^{\underline{5}}$ would equal 10 * 9 * 8 * 7 * 6 = 30240. Nothing important changes if the base `n` is negative. For example, the falling power $(-4)^{\underline{5}}$ is computed the exact same way as -4 * -5 * -6 * -7 * -8 = -6720.

This method should compute and return the falling power $n^{\underline{k}}$ where *n* can be any integer, and *k* can be any nonnegative integer. (Analogous to ordinary powers, $n^{\underline{0}} = 1$ for any *n*.) The automated tester is designed so that your method does not need to worry about potential integer overflow.

```
public static int[] everyOther(int[] arr)
```

Given an integer array `arr`, create and return a new array that contains precisely the elements in the even-numbered positions in the array `arr`. Make sure that your method works correctly for arrays of both odd and even lengths, and for arrays that contain zero or only one element. The length of the result array that you return must be exactly right so that there are no extra zeros at the end of the array.

```
public static int[][] createZigZag(int rows, int cols, int start)
```

This method creates and returns a new two-dimensional integer array, which in Java is really just a one-dimensional array whose elements are one-dimensional arrays of type `int[]`. The returned array must have the correct number of `rows` that each have exactly `cols` columns. This array must contain the numbers `start`, `start + 1`, ..., `start + (rows * cols - 1)` in its rows in order, except that the elements in each odd-numbered row must be listed in descending order.

For example, when called with `rows = 4`, `cols = 5` and `start = 4`, this method should create and return the two-dimensional array whose contents are

```
4     5     6     7     8
13    12    11    10    9
14    15    16    17    18
23    22    21    20    19
```

when displayed in the traditional matrix form that is more readable for the human than the more realistic form of a one-dimensional array whose elements are one-dimensional arrays of rows.

```
public static int countInversions(int[] arr)
```

Inside an array `a`, an **inversion** is a pair of positions `i` and `j` inside the array that satisfy simultaneously both `i < j` and `a[i] > a[j]`. In combinatorics, the inversion count inside an array is a rough measure how "out of order" that array is. If the array is sorted in ascending order, it has zero inversions, whereas an n-element array that is sorted in reverse order has $n(n-1)/2$ inversions, the largest number possible. This method should count the inversions inside the given array `arr`, and return that count. (As you should always do when writing methods that operate on arrays, make sure that your method works correctly for arrays of any length, including the important special cases of zero and one.)

Once you have written all four methods, you can download and add the above **JUnit test** class P2JTest.java to be placed inside the same BlueJ project. In the BlueJ project display window, the JUnit test class show up as a green box, as opposed to the usual yellow box like the ordinary classes. **The JUnit test class cannot be compiled until your class contains all four methods exactly as they are specified.** If you want to test one method without having to first write also the other three, you can implement the other three methods as one-liners with some placeholder return statement that returns zero or some other dummy value. These methods will of course fail their respective tests, but you can test the one method that you have written, and once that is done, move on to replace the placeholder body of later method with the actual body.

Once successfully compiled, you can right-click the JUnit test class to run either any one test, or all tests at once. The methods that receive a green checkmark have passed the tester and are complete. A red checkmark means that your method returned a wrong answer at some point, whereas a black X-mark means that your method crashed at some point and threw an exception.

Because these JUnit test methods, as implemented by your instructor, work by calling your method with a large number of pseudo-randomly generated test cases, and compute a checksum of the results returned by your method to be compared to the checksum produced by the instructor's private model answer, it is impossible to point out precisely which particular test cases are different and failing for your method. You should write your own small tests to find the errors in your code.

Lab 0(B): Strings

JUnit: P2J2Test.java

Create a new class named `P2J2` inside the very same BlueJ project as you placed your `P2J` class in the previous lab. Inside this new class `P2J2`, write the following four methods.

```
public static String removeDuplicates(String text)
```

Given a `text` string, create and return a new string that is otherwise the same but every run of equal consecutive characters has been turned into a single character. For example, given the arguments `"Kokkarinen"` and `"aaaabbxxxxaaxa"`, this method would return `"Kokarinen"` and `"abxaxa"`, respectively. Note that only the consecutive duplicate occurrences of the same character are eliminated, but the later occurrences of the same character remain in the result as long as there was some other character between the previous occurrence.

```
public static String uniqueCharacters(String text)
```

Given a `text` string, create and return a new string that contains each character only once, with the characters given in order in which they appear in the original string. For example, given the arguments `"Kokkarinen"` and `"aaaabbxxxxaaxa"`, this method would return `"Kokarine"` and `"abx"`, respectively.

You can solve this problem with two nested loops, the outer loop looping through the positions of the `text`, and the inner loop looping through all previous positions looking for a previous occurrence of that character. Or you can be much more efficient, and use a `HashSet<Character>` to remember which characters you have already seen so that you can determine in O(1) time whether to add the next character into the result.

```
public static int countSafeSquaresRooks(int n, boolean[][] rooks)
```

Some number of rooks have been placed on some squares of a generalized `n`-by-`n` chessboard. The two-dimensional array `rooks` of boolean truth values tells you which squares contain a rook. (This array is guaranteed to be exactly `n`-by-`n` in size.) This method should count how many remaining squares are safe from these rooks, that is, do not contain any rooks in the same row or column, and return that count.

```
public static int recaman(int n)
```

Compute and return *n*:th term of the Recamán's sequence, as defined on Wolfram Mathworld, starting from the term $a_1 = 1$. See the definition of this sequence on that page. For example, when called with $n = 7$, this method would return 20, and when called with $n = 19$, return 62. (More values are listed at OEIS sequence A005132.)

To make your function fast and efficient even when computing the sequence element for large values of *n*, you should use a sufficiently large `boolean[]` (size `10*n` is certainly enough, and yet needs only 10 bits per each number) to keep track of which integer values are already part of the generated sequence, so that you can generate each element in constant time instead of having to loop through the entire previously generated sequence like some "Shlemiel" would do.

Lab 0(C): Putting Details Together

JUnit: P2J3Test.java

One last batch of transitional problems taken from the instructor's Python graded labs. The automated tester for these labs uses the `warandpeace.txt` text file, so make sure that this file has been properly downloaded into your course labs project folder.

```
public static void reverseAscendingSubarrays(int[] items)
```

Rearrange the elements of the given an array of integers **in place** (that is, do not create and return a new array) so that the elements of every **maximal strictly ascending subarray** are reversed. For example, given the array `{ 5, 7, 10, 4, 2, 7, 8, 1, 3 }` (the colours indicate the ascending subarrays and are not actually part of the argument), after executing this method, the elements of the array would be `{ 10, 7, 5, 4, 8, 7, 2, 3, 1 }`. Given the array `{ 5, 4, 3, 2, 1 }`, it would become `{ 5, 4, 3, 2, 1 }` since each element by itself is a maximal ascending subarray of length one.

```
public static String pancakeScramble(String text)
```

This nifty little problem is taken from the excellent Wolfram Challenges problem site where you can see examples of what the result should be for various arguments. Given a `text` string, construct a new string by reversing its first two characters, then reversing the first three characters of that, and so on, until the last round where you reverse your entire current string.

This problem is an exercise in Java string manipulation. For some mysterious reason, the Java `String` type does not come with a `reverse` method. The canonical way to reverse a Java string `str` is to first convert it to mutable `StringBuilder`, reverse that one, and convert the result back to a string, that is, `str = new StringBuilder(str).reverse().toString()`.

```
public static String reverseVowels(String text)
```

Given a `text` string, create and return a new string of same length where all vowels have been reversed, and all other characters are kept as they were. For simplicity, in this problem only the characters `aeiouAEIOU` are considered vowels, and `y` is never a vowel. For example, given the text string `"computer science"`, this method would return the new string `"cempetir sceunco"`.

Furthermore, to make this problem more interesting and the result look prettier, this method **must maintain the capitalization of vowels** based on the vowel character that was originally in the position that each new vowel character is moved into. For example, `"Ilkka Markus"` should become `"Ulkka Markis"` instead of `"ulkka MarkIs"`. Use the handy utility methods in the

`Character` wrapper class to determine whether some particular character is upper- or lowercase, and to convert some character to upper- or lowercase as needed.

```
public static boolean subsetSum(int[] items, int n, int goal)
```

The famous subset sum problem, interesting and important in theoretical computer science, asks whether the first `n` elements of the parameter array `items`, guaranteed to contain only positive integers in ascending sorted order, contains some subset of elements that together add up exactly to `goal`. Each element of the array can be used at most once as part of this sum. For example, in the array `{2, 4, 5, 8, 11, 13, 17}`, the subset `{4, 8, 11}` adds up to the goal `23`.

This problem is easiest to solved as **branching recursion** whose base cases are `true` when `goal == 0` and `false` when `n == 0`. Otherwise, the first `n` elements of `items` contain such a subset if either the first `n-1` elements contain such a subset, or if the first `n-1` elements contain a subset that adds up to `goal-items[n-1]`. However, note that since each recursive call generates two more recursive calls, the resulting recursion will take exponential time with respect to the length of the array. (Interested students might want to try to think up ways to cut down this running time with various little optimizations.)

Lab 1: Polynomial Data Type: Basics

JUnit: PolynomialTestOne.java

After learning the basics of Java language, we may proceed to designing our own data types as **classes**, and writing the operations on these data types as **methods** that the outside users of our data type can then call to get the job done.

In spirit of the Fraction example class seen in the first lecture, your first task in this course is to implement the class `Polynomial` whose objects represent *polynomials* of variable *x* with integer coefficients, and their basic mathematical operations. If your math skills on polynomials have become a bit rusty since you last had to use them somewhere back in high school, check out the page "Polynomials" in the "College Algebra" section of "Paul's Online Math Notes", the best and still very concise online resource for college level algebra and calculus that I know of.

As is the good programming style unless there exist good reasons to do otherwise, this class will be intentionally designed to be *immutable* so that `Polynomial` objects cannot change their internal state after they have been constructed. Immutability has many advantages in programming, even though those advantages might not be fully evident yet. The `public` interface of `Polynomial` should consist of the following instance methods.

```
@Override public String toString()
```

Implement this method as your very first step to return some kind meaningful, human readable `String` representation of `this` instance of `Polynomial`. This method is not subject to testing by the JUnit testers, so you can freely choose for yourself the exact textual representation that you'd like this method to produce. Having this method implemented properly will become **immensely** useful for debugging all the remaining methods that you will write inside `Polynomial` class!

```
public Polynomial(int[] coefficients)
```

The *constructor* that receives as argument the array of *coefficients* that define the polynomial. For a polynomial of degree *n*, the array `coefficients` contains exactly *n* + 1 elements so that the coefficient of the term of order *k* is in the element `coefficients[k]`. For example, the polynomial $5x^3 - 7x + 42$ that will be used as example in all of the following methods would be represented as the coefficient array `{42, -7, 0, 5}`.

Terms missing from inside the polynomial are represented by having a zero coefficient in that position. However, the **coefficient of the highest term of every polynomial should always be nonzero**, unless the polynomial itself is identically zero. If this constructor is given as argument a coefficient array whose highest terms are zeroes, it should simply ignore those zero coefficients. For example, if given the coefficient array `{-1, 2, 0, 0, 0}`, the resulting polynomial would have the degree of only one, as if that coefficient array had been `{-1, 2}` without those leading zeros.

To guarantee that the `Polynomial` class is immutable so that no outside code can ever change the internal state of an object after its construction (well, at least not without resorting to underhanded Java tricks such as *reflection*), the constructor should not assign only the reference to the `coefficients` array to the `private` field of coefficients, but it absolutely positively **must create a separate but identical *defensive copy* of the argument array, and store that defensive copy instead**. This technique ensures that the stored coefficients of the polynomial do not change if some outsider later changes the contents of the shared `coefficients` array that was passed as the constructor argument.

```
public int getDegree()
```

Returns the degree of `this` polynomial, that is, the exponent of its highest order term. For example, the previous polynomial has degree 3. Constant polynomials have a degree of zero.

```
public int getCoefficient(int k)
```

Returns the coefficient for the term of order *k*. For example, the term of order 3 of the previous polynomial equals 5, and the term of order 0 equals 42. This method should work correctly even when *k* is negative or greater than the actual degree of the polynomial, and simply return zero in such cases of nonexistent terms. (This policy will also make some methods of Lab 2 much easier to implement.)

`public long evaluate(int x)`

Evaluates the polynomial using the value `x` for the unknown symbolic variable of the polynomial. For example, when called with `x = 2` for the previous example polynomial, this method would return `68`. To keep this simple, your method does not have to worry about potential *integer overflows*, but can assume that the final and intermediate results of this computation will always stay within the range of the primitive data type `long`.

Lab 2: Polynomial Data Type: Arithmetic

JUnit: PolynomialTestTwo.java

The second lab continues with the `Polynomial` class from the first lab by adding new methods for *polynomial arithmetic* to its source code. (There is no inheritance or polymorphism taking place yet in this lab.) Since the class `Polynomial` is designed to be immutable, none of the following methods should modify the objects `this` or `other` in any way, but return the result of that arithmetic operation as a brand new `Polynomial` object created inside that method.

`public Polynomial add(Polynomial other)`

Creates and returns a new `Polynomial` object that represents the result of *polynomial addition* of the two polynomials `this` and `other`. This method should not modify `this` or `other` polynomial in any way. Make sure that just like with the constructor, the coefficient of the highest term of the result is nonzero, so that adding the two polynomials $5x^{10} - x^2 + 3x$ and $-5x^{10} + 7$, each having a degree 10, produces the result $-x^2 + 3x + 7$ that has a degree of only 2 instead of 10.

`public Polynomial multiply(Polynomial other)`

Creates and returns a new `Polynomial` object that represents the result of *polynomial multiplication* of the two polynomials `this` and `other`. Polynomial multiplication works by multiplying all possible pairs of terms between the two polynomials and adding them together, combining terms of equal degree together into a single term.

Lab 3: Extending an Existing Class

JUnit: AccessCountArrayListTest.java

In the third lab, you get to practice using *class inheritance* to create your own custom subclass versions of existing classes in Java with new functionality that did not exist in the original superclass. Your third task in this course is to use inheritance to create your own custom subclass `AccessCountArrayList<E>` that `extends` the good old workhorse `ArrayList<E>` from the Java Collection Framework. This subclass should maintain in an internal data field an `int` count of how

many times the methods `get` and `set` have been called. (The same `int` counter keeps the simultaneous count for both of these methods together.)

You should override the inherited `get` and `set` methods so that both of these methods first increment the access counter, and only then call the superclass version of that same method (use the prefix `super` in the method call to make this happen), returning whatever result that superclass version returned. In addition to these overridden methods inherited from the superclass, your class should define the following two brand new methods:

```
public int getAccessCount()
```

Returns the current count of how many times the `get` and `set` methods have been called for `this` object.

```
public void resetCount()
```

Resets the access count field of `this` object back to zero.

Lab 4: Polynomial Data Type: Comparisons

JUnit: PolynomialTestThree.java

In this fourth lab, we continue modifying the source code for the `Polynomial` class from the first two labs to allow *equality* and *ordering* comparisons to take place between objects of that type. Modify the class definition so that this class `implements Comparable<Polynomial>`. Then write the following methods to implement the equality and ordering comparisons.

```
@Override public boolean equals(Object other)
```

Returns `true` if the `other` object is also a `Polynomial` of the exact same degree as `this`, and that the coefficients of `this` and `other` polynomial are pairwise equal. If the `other` object is anything else, this method should return `false`.

(To save you some time, you can actually implement this method after implementing the method `compareTo` below, since once that method is available, the logic of equality checking will be a trivial one-liner after the `instanceof` check.)

```
@Override public int hashCode()
```

Whenever you override the `equals` method in any subclass, you should also override the `hashCode` method to ensure that two objects that are considered equal by the `equals` method will also have equal integer hash codes. This method computes and returns the *hash code* of this polynomial, used

to store and find this object inside some instance of `HashSet<Polynomial>`, or some other *hash table* based data structure.

You get to choose for yourself the hash function that you implement, but like all hash functions, the result should depend on the degree and all of the coefficients of your polynomial. The hash function absolutely **must** satisfy the contract that whenever `p1.equals(p2)` holds for two `Polynomial` objects, then also `p1.hashCode() == p2.hashCode()` holds for them.

Of course, since this entire problem is so common and it seems silly to force everyone to reinvent the same wheel again, these days you can use the method `hash` in the `java.util.Objects` utility class to compute a good hash value for your coefficients. (The method `hash` in the class `Objects` is a **vararg method** meaning that it can accept any number of arguments, or an array, so you can simply pass your `coefficients` array to that method, assuming that you have made sure that you don't store any leading zero coefficients in this array.)

```
public int compareTo(Polynomial other)
```

Implements the *ordering comparison* between `this` and `other` polynomial, as required by the interface `Comparable<Polynomial>`, allowing the instances of `Polynomial` to be *sorted* or stored inside some instance of `TreeSet<Polynomial>`. This method returns `+1` if `this` is greater than `other`, `-1` if `other` is greater than `this`, and `0` if both polynomials are equal in the sense of the `equals` method.

A total ordering relation between polynomials can be defined by many rules. Here we shall use an ordering rule that says that **any polynomial of a higher degree is automatically greater than any polynomial of a lower degree**, regardless of their coefficients. For two polynomials whose degrees are equal, the result of the order comparison is determined by **the highest-order term for which the coefficients of the polynomials differ**, so that the polynomial with a larger such coefficient is considered to be greater in this ordering.

Be careful to ensure that this method ignores the leading zeros of high order terms if you have them inside your polynomial coefficient array, and that **the ordering comparison criterion is precisely the one defined in the previous paragraph**. Otherwise the automated tester will reject your code, even if your code happened to define some other perfectly legal ordering relation from the infinitely many possible ordering relations!

Lab 5: Introduction to Swing

Historically, GUI component programming was the first "killer app" of object oriented programming that caused this entire paradigm to hit into the mainstream after two decades of purely theoretical academic research. In this lab, your task is to create a custom Swing GUI component `Head` that extends `JPanel`. To practice working with Java graphics, this component should display a simple

human head that, to also practice the Swing *event handling* mechanism, reacts to the mouse cursor entering and exiting the surface of that component. You should copy-paste a bunch of boilerplate code from the example class `ShapePanel` into this class.

Your class should contain one field `private boolean mouseInside` that is used to remember whether the mouse cursor is currently over your component, and the following methods:

```
public Head()
```

The constructor of the class should first set the preferred size of this component to be 500-by-500 pixels, and then give this component a decorative raised bevel border using the utility method `BorderFactory.createBevelBorder`. Next, this constructor adds a `MouseListener` to this component, this event listener object constructed from an inner class `MyMouseListener` that `extends MouseAdapter`. Override both methods `mouseEntered` and `mouseExited` inside your event listener class to first set the value of the field `mouseInside` accordingly and then call `repaint`.

```
@Override public void paintComponent(Graphics g)
```

Renders some kind of image of a simple human head on the surface of this component. This is not an art class so this head does not need to look fancy, but a couple of simple rectangles and ellipses suffice. (Of course, interested students of more artistic bent might want to check out more complicated shapes from the package `java.awt.geom`, or perhaps `Image` objects read from some GIF or JPEG files, to generate a prettier image.)

If the value of the field `mouseInside` equals `true`, the eyes of this head should be drawn to be open, whereas if `mouseInside` equals `false`, the eyes should be drawn to be closed. (As long as the eyes are somehow noticeably visually different based on the mouse entering and exiting the component surface, that is enough for this lab.)

To admire your reactive Swing component on the screen, write a separate `HeadMain` class that contains a `main` method that creates a `JFrame` that contains four separate `Head` components arranged in a neat 2-by-2 grid using the `GridLayout` layout manager. Move your cursor from one component to other to watch the eyes open and close with these mouse movements.

Lab 6: Processing Text Files

JUnit: WordCountTest.java

This lab lets you try out reading in text data from an instance of `BufferedReader` and performing computations on that data. To practice working inside a small but entirely proper object-oriented *framework*, we design an entire class hierarchy that reads in text one line at the time and performs

some operation for each line. The subclasses can then decide what exactly that operation is by overriding the *template methods* used by this algorithm.

To allow this processing to return a result of arbitrary type that can be freely chosen by the users of this class, the abstract superclass of this little framework is also defined to be *generic*. Start by creating the class `public abstract class FileProcessor<R>` that defines the following three `abstract` methods:

```
protected abstract void startFile();
protected abstract void processLine(String line);
protected abstract R endFile();
```

and one concrete `final` method that is therefore guaranteed to be the same for all the future subclasses in this entire class hierarchy:

```
public final R processFile(BufferedReader in) throws IOException
```

This method should first call the method `startFile`. Next, it reads all the lines of text coming from `in` one line at the time in some kind of suitable loop, and calls the method `processLine` passing it as argument each line that it reads in. Once all incoming lines have been read and processed, this method should finish up by calling the method `endFile`, and return whatever the method `endFile` returned.

This abstract superclass defines a template for a class that performs some computation for a text file that is processed one line at the time, returning a result whose type `R` can be freely chosen by the concrete subclasses of this class. Different subclasses can now override the three template methods `startFile`, `processLine` and `endFile` methods in different ways to implement different computations on text files.

As the first application of this mini-framework (but it is a framework by definition, **since you will be writing methods for this framework to call, instead of the framework offering methods for you to call**), you will emulate the core functionality of the Unix command line tool wc that counts how many characters, words and lines the given file contains. Those of you who have taken the Unix and C programming course CCPS 393 should already recognize this handy little text processing tool, but even if you have not yet taken that course, you can still implement this tool based on the following specification of its behaviour.

Create a class `WordCount` that `extends FileProcessor<List<Integer>>`. This class should contain three integer fields to keep track of these three counts, and the following methods:

```
protected void startFile()
```

Initializes the character, word and line counts to zero.

```
protected void processLine(String line)
```

Increments the character, word and line counts appropriately. In the given `line`, every character increments the count regardless whether it is a whitespace character or not. To properly count the words in the given line, count the non-whitespace characters that either begin that `line`, or whose previous character on that `line` is some whitespace character. You must use the utility method `Character.isWhitespace` to test whether some character is a whitespace character, since the Unicode standard defines quite a lot more whitespace characters than most people can even name.

```
protected List<Integer> endFile()
```

Creates and returns a new `List<Integer>` instance (you can choose the concrete subtype of this result object for yourself) that contains exactly three elements; the character, word and line counts, in this order.

Lab 7: Concurrency in Animation

This seventh lab lets you create a Swing component that displays a real-time animation using Java concurrency to execute an *animation thread* that runs independently of what the human user happens to be doing with that component. This thread will animate a classic *particle field* where a swarm of thousands of independent little *particles* buzzes randomly around the component.

(This same basic architecture could then, with surprisingly little additional modification, be used to implement some real-time game, with this animation thread moving the game entities according to the rules of the game 50 frames per second, and the event listeners giving orders to the game entity that happens to represent the human player. As it seems to be in all walks of life, most things that you expect to be complex and difficult usually turn out to be surprisingly much simpler than you would have assumed, whereas the things that you expected to be simple and easy often turn out to be dizzyingly complex once you crack them open and try to implement them yourself without any hand waving!)

First, create a class `Particle` whose instances represent individual particles that will randomly around the two-dimensional plane. Each particle should remember its *x*- and *y*-coordinates on the two-dimensional plane and its heading as an angle expressed as **radians**, stored in three data fields of the type `double`. This class should also have a random number generator shared between all objects, and a shared field `BUZZY` that defines how random the motion is.

```
private static final Random rng = new Random();
private static final double BUZZY = 0.7;
```

The class should then have the following methods:

```
public Particle(int width, int height)
```

The constructor that places `this` particle in random coordinates inside the box whose possible values for the *x*-coordinate range from 0 to `width`, and for the *y*-coordinate from 0 to `height`. The initial heading is taken from the expression `Math.PI * 2 * rng.nextDouble()`.

```
public double getX()
public double getY()
```

The accessor methods for the current *x*- and *y*-coordinates of `this` particle.

```
public void move()
```

Updates the value of `x` by adding `Math.cos(heading)` to it, and updates the value of `y` by adding `Math.sin(heading)` to it. After that, the `heading` is updated by adding the value of the expression `rng.nextGaussian() * BUZZY` to it.

Having completed the class to represent individual particles, write a class `ParticleField` that extends `javax.swing.JPanel`. The instances of this class represent an entire field of random particles. This class should have the following `private` instance fields.

```
private boolean running = true;
private java.util.List<Particle> particles =
   new java.util.ArrayList<Particle>();
```

The class should have the following `public` methods:

```
public ParticleField(int n, int width, int height)
```

The constructor that first sets the preferred size of this component to be `width`-by-`height`. Next, it creates `n` separate instances of `Particle` and places them in the `particles` list.

Having initialized the individual particles, this constructor should create and launch one new `Thread` using a `Runnable` argument whose method `run` consists of one `while(running)` loop. The body of this loop should first `sleep` for 20 milliseconds. After waking up from its sleep, it should loop through all the `particles` and call the method `move()` for each of them. Then call `repaint()` and go to the next round of the while-loop.

```
@Override public void paintComponent(Graphics g)
```

Renders `this` component by looping through particles and rendering each one of them as a 3-by-3 pixel rectangle to its current *x*- and *y*-coordinates.

```
public void terminate()
```

Sets the field `running` of `this` component to be `false`, causing the animation thread to terminate in the near future.

To admire the literal and metaphorical buzz of your particle swarm, write another class `ParticleMain` that contains a `main` method that creates one `JFrame` instance that contains a `ParticleField` of size 800-by-800 that contains 2,000 instances of `Particle`. Using the `main` method of the example class `SpaceFiller` as a model of how to do this, attach a `WindowListener` to the `JFrame` so that the listener's method `windowClosing` first calls the method `terminate` of the `ParticleField` instance shown inside the `JFrame` before disposing that `JFrame` itself.

Try out the effect that different values between 0.0 and 10.0 for `BUZZY` have to the motion of your particles. When `BUZZY` is small, the motion of each particle tends the maintain its current direction, whereas larger values of `BUZZY` make the motion turn into random Brownian motion.

(Particle systems are often used in games to create interesting animations of phenomena such as explosions or fire that would be otherwise difficult to simulate and render. Giving particles more intelligence and mutual interaction such as making some particles follow or avoid other particles can produce surprisingly natural and lifelike emergent animations.)

Lab 8: Processing Text Files, Part II

JUnit: TailTest.java

In this lab, we return to the `FileProcessor<R>` framework from Lab 6 for writing tools that process text files one line at the time. This time this framework will be used to implement another Unix command line tool tail that extracts the last `n` lines of its input and discards the rest. Write a class `Tail` that extends `FileProcessor<List<String>>` and has the following methods:

```
public Tail(int n)
```

The constructor that stores its argument `n`, the number of last lines to return as the result, into a private data field. In this lab, you have to choose for yourself what instance fields you need to define in your class to make the required methods work.

```
@Override public void startFile()
```

Start processing a new file from the beginning. Nothing to do here, really.

```
@Override public void processLine(String line)
```

Process the current `line`. Do something intelligent here. Be especially careful not to be a "Shlemiel" so that your logic of processing each line always has to loop through all of the previous lines that you have collected and stored so far. To avoid being a "Shlemiel", note that the `List<String>` instance that you create and return does not necessarily have to be specifically an `ArrayList<String>`, but perhaps some other subtype of `List<String>` that allows O(1) mutator operations at both of its ends would be far more appropriate...

```
@Override public List<String> endFile()
```

Returns a `List<String>` instance that contains precisely the `n` most recent lines that were given as arguments to the method `processLine` in the same order that they were originally read in. If fewer than `n` lines have been given to the method `processLine` since the most recent call to the method `startFile`, this list should contain as many lines as have been given.

Lab 9: Swinging Some Curves

In this lab you write one last graphical Swing component that displays a Lissajous curve on its surface, so that the user can control the parameters *a*, *b* and *delta* that define the shape of this curve by entering their values into three `JTextField` components placed inside this component. Write a class `Lissajous extends JPanel`, and the following methods in it:

```
public Lissajous(int size)
```

The constructor that sets the preferred size of this component to be `size`-by-`size` pixels. Then, three instances of `JTextField` are created and added inside `this` component. Initialize these text fields with values 2, 3 and 0.5, with some extra spaces added to these strings so that these text fields have a decent size for the user to enter other numbers. Add an `ActionListener` to each of the three text fields whose method `actionPerformed` simply calls `repaint` for this component.

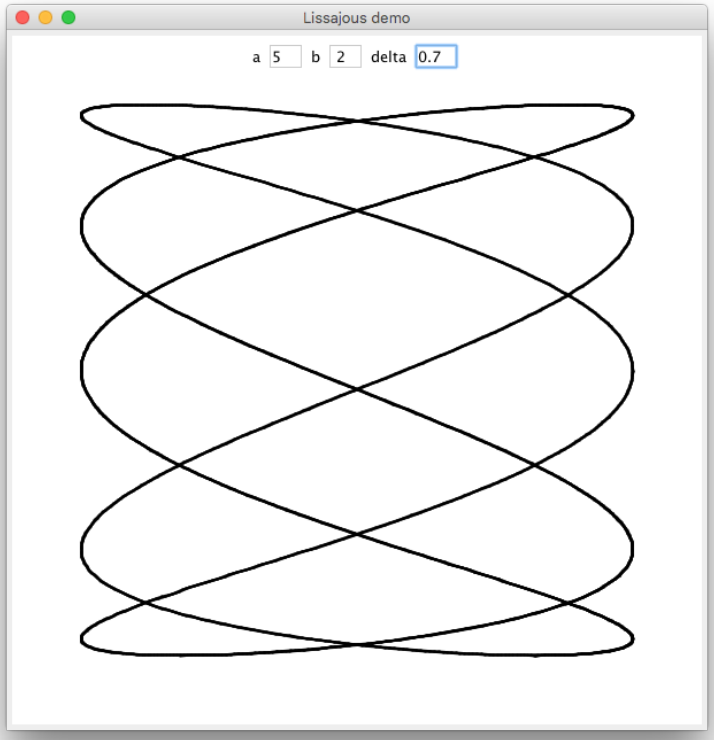
```
@Override public void paintComponent(Graphics g)
```

Renders the Lissajous curve on the component surface, using the current values for *a*, *b* and *delta* that it first reads from the previous three text fields. This method should consist of a for-loop whose loop counter `double t` goes through the values from `0` to `(a + b) * Math.PI` using some suitably small increment. In the body of the loop, compute the coordinates `x` and `y` of the current point using the formulas

```
x = size/2 + 2*size/5 * Math.sin(a * t + delta);
y = size/2 + 2*size/5 * Math.cos(b * t);
```

and draw a line segment from the current point to the previous point.

Again, to get to admire your Lissajous curve and try out the effect of different values of *a*, *b* and *delta* on the shape of the curve as if you were trapped inside the lair of a mad scientist of some 1970's dystopian science fiction movie, create a separate class `LissajousMain` whose `main` method creates a `JFrame` that contains your `Lissajous` component. The end result should look somewhat like this:



Motivated students can take on as an extra challenge to make the displayed image look smoother by eliminating some visual jagginess, and possibly even make this rendering look more artistic in other ways. (For example, instead of subdividing the curve into linear line segments, you could rather subdivide it into much smoother cubic curves that connect seamlessly at the point and direction vector that the previous curve left off...)

Lab 10: Computation Streams

JUnit: StreamExercisesTest.java

This last lab teaches you to think and solve problems in the *functional programming* framework of the *Java 8 computation streams*. Therefore in this lab, you are **absolutely forbidden** to use any conditional statements (either `if` or `switch`), loops (either `for`, `while` or `do-while`) or even

recursion. Instead, all computation **must** be implemented using **only Java computation streams and their operations!**

In this lab, we shall also check out the **Java NIO framework** for better file operations than those offered in the old package `java.io` and the class `File`. Your methods receive a `Path` as an argument, referring to the text file in your file system whose contents your methods will then process with computation streams. Write both of the following methods inside a new class named `StreamExercises`.

```
public int countLines(Path path, int thres) throws IOException
```

This method should first use the utility method `Files.lines` to convert the given `path` into an instance of `Stream<String>` that produces the lines of this text file as a stream of strings, one line at the time. The method should then use `filter` to keep only those whose `length` is greater or equal to the threshold value `thres`, and in the end, return a count how many such lines the file contains.

```
public List<String> collectWords(Path path) throws IOException
```

This method should also use the same utility method `Files.lines` to first turn its parameter `path` into a `Stream<String>`. Each line should be converted to lowercase and broken down to individual words that are passed down the stream as separate `String` objects (the stream operation `flatMap` will be handy here). Split each line into its individual words using the separator regex `"[^a-z]+"` for the method `split` in `String`. In the next stages of the computation stream, discard all empty words, and `sort` the remaining words in alphabetical order. Multiple consecutive occurrences of the same word should then be removed (use the stream operation `distinct`, or if you want to do this yourself the hard way as an exercise, write a custom `Predicate<String>` subclass whose method `test` accepts its argument only if it was the first one or distinct from the previous argument), and to wrap up this computation stream, `collect`ed into a `List<String>` that is returned as the final answer.