Sarajevo School of Science and Technology
CS/IS 160 Programming and Problem Solving II
June 2015

Final Exam Practice Solutions

Product of diagonal entries of a square matrix.

Implement a static method `diagProd`, which returns the product of the diagonal entries of a square matrix. For example, if the input matrix is given as

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

then the sum of its diagonal entries is $1 \times 5 \times 9 = 45$.

Use the following method prototype, where `matrix` is an input integer square matrix:

```
public static int diagProd( int[][] matrix )
```

Solution: An implementation of the method is shown below.

```
public static int diagProd( int[][] matrix ) {

      int prod = 1;

      for( int i=0; i < matrix.length; i++ ) {
            prod *= matrix[i][i];
      }

      return prod;
}
```

Filtered counter for a square matrix.

Implement a static method `countGreaterThan`, which returns the number of elements of a square matrix that are greater than an input threshold value. For example, if the input matrix is given as

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

and the threshold value is 5, then the returned count is 4 (there are four numbers in the matrix that are greater than 5).

Use the following method prototype, where `matrix` is an input integer square matrix and `val` is the threshold value:

```
public static int countGreaterThan( int[][] matrix, int val )
```

Solution: An implementation of the method is shown below.

```
public static int countGreaterThan( int[][] matrix, int val ) {

      int count = 0;

      for( int i=0; i < matrix.length; i++ ) {
            for( int j=0; j < matrix[i].length; j++) {
                  if( matrix[i][j] > val ) {
                        count++;
                  }
            }
      }

      return count;
}
```

Downsample an array.

Downsampling by a factor of *n* is a digital signal processing (DSP) technique in which we take an input array and return a smaller array created by sampling every *n*th element, starting with the first one.

Implement a static method `downsample`, for which the inputs are an array and a downsample factor, while the output is the downsampled array. For example, if the input array is (1, 2, 1, 0, 5, 6, 7) and the factor is 3, then the output array is (1, 0, 7).

Use the following method prototype, where `array` is an input integer array and `n` is the downsample factor:

```
public static int[] downsample( int[] array, int n )
```

Solution: An implementation of the method with result array allocation is shown below.

```
public static int[] downsample( int[] array, int n ) {

      int[] result = new int[array.length/n + array.length % n];

      for( int i=0, j=0; j < array.length; i++, j+=n ) {
            result[i] = array[j];
      }

      return result;
}
```

Equal arrays.

Write a static method `equalArrays` which tests if two arrays of integers are identical, that is, if they contain the same elements in the same order. Use the following method prototype, where `a` is the first array and `b` is the second array:

```
public static boolean equalArrays( int[] a, int[] b )
```

Solution: An implementation of the method using a for-loop is shown below.

```
public static boolean equalArrays( int[] a, int[] b ) {

      if( a.length != b.length ) {
            return false;
      }

      for( int i=0; i < a.length; i++ ) {
            if( a[i] != b[i] ) {
                  return false;
            }
      }

      return true;
}
```

A very simple solution using the `equals` method from the `Arrays` class is also possible.

```
public static boolean equalArrays(int[] a, int[] b) {

      return Arrays.equals(a, b);
}
```

Equal lists.

Write a static method `equalLists` which tests if two lists of integers are identical, that is, if they contain the same elements in the same order. Use the following method prototype, where `a` is the first list and `b` is the second one:

```
public static boolean equalLists( List<Integer> a, List<Integer> b )
```

Solution: A solution for the lists is identical to the one for arrays, except the difference in element access and list size syntax.

```
public static boolean equalLists(List<Integer> a, List<Integer> b) {

      if( a.size() != b.size() ) {
            return false;
      }

      for( int i=0; i < a.size(); i++ ) {
```

```
            if( a.get(i) != b.get(i) ) {
                  return false;
            }
      }

      return true;
}
```

A simple solution is also possible using the `equals` instance method from the lists.

```
public static boolean equalLists(List<Integer> a, List<Integer> b) {

      return a.equals(b);
}
```

String max run.

Write a static method `maxRun`, which given an input string returns the length of the longest *run* in the string. A *run* is defined as a series of zero or more adjacent characters that are the same. So the max run of "xxyyyz" is 3, and the max run of "xyz" is 1. Note that the string instance method `charAt(int index)` can be used to obtain a character located at the index location in the string.

Use the following method prototype:

```
public static int maxRun( String str )
```

Solution: An implementation of the method using a for-loop and accessing characters from the string is shown below. At each iteration, we check if the current run is greater then the maximum run.

```
      public static int maxRun( String str ) {

            if(str.length() == 0) {
                  return 0;
            }

            int run = 1;
            int maxRun = 1;

            for( int i=0; i < str.length()-1; i++ ) {

                  if(str.charAt(i) == str.charAt(i+1)) {
                        run++;

                        if(run > maxRun) {
                              maxRun = run;
                        }
                  }
                  else {
                        run = 1;
                  }
```

```
        }

        return maxRun;
    }
```

Shuffle a string list of names.

Write a static method `shuffleNames` that takes a list of String objects and rearranges them randomly inside the list. Use the following method prototype:

```
public static void shuffleNames( List<String> names )
```

Solution: A sample solution is shown below using a loop that does a random swap of two characters at each iteration. We can repeat this swapping as many times as we want (or until sufficient shuffle randomness is achieved).

```
public static void shuffleNames( List<String> names ) {

        int m, n;
        String temp;

        Random rand = new Random();

        for(int i=0; i < names.size()*10; i++) {

                m = rand.nextInt(names.size());
                n = rand.nextInt(names.size());

                temp = names.get(m);
                names.set(m, names.get(n));
                names.set(n, temp);
        }
}
```

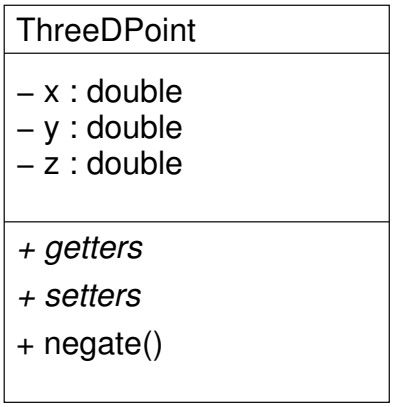
A very simple solution using the `shuffle` method from the `Collections` class is also possible.

```
public static void shuffleNames( List<String> names ) {

        Collections.shuffle(names);
}
```

Point in 3D space class.

Implement a simple `ThreeDPoint` class that models points in 3D vector space (each point has three coordinates `x`, `y`, and `z`, stored as private double variables).

The UML diagram for the class is shown below.



a) Implement a default constructor that sets coordinates to zeros.

b) Implement the getter and setter method for the `x` variable.

c) Implement the public method `negate`, which flips the sign of each vector coordinate.

d) Implement the public instance method `toString` to return the following string representation of the `ThreeDPoint` object "(x, y, z)", where x, y, and z are the values of the vector coordinates.

Use the following instance method prototype:

```
public String toString()
```

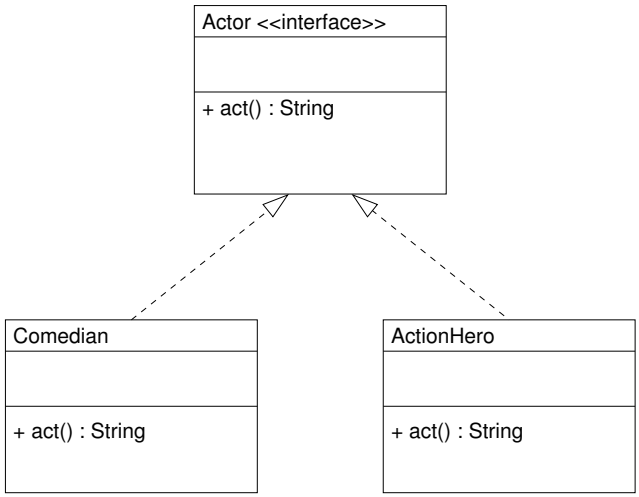
Solution: An implementation of the `ThreeDPoint` class is shown below.

```
public class ThreeDPoint {

	private double x, y, z;

	// part (a) default constructor

	public ThreeDPoint() {
		x = 0.0;
		y = 0.0;
		z = 0.0;
	}

	// part (b) getter and setter for x

	public double getX() {
		return x;
	}
```

```
	public void setX(double x) {
		this.x = x;
	}

	// part (c) negate instance method

	public void negate() {
		x = -x;
		y = -y;
		z = -z;
	}

	// part (d) toString instance method

	public String toString() {
		return "(" + x + "," + y + "," + z + ")";
	}

}
```

Polymorphism with Actor classes.

We have the class hierarchy shown below and implemented as practice for Test 3. Here, the `Actor` is an interface and the `Comedian` and `ActionHero` classes implement the interface. The `act` method for the `Comedian` class should return the string "I'm Austin Powers," while the `ActionHero` class should return "My name is Bond, James Bond."



Write a main method that creates an `Actor[]` array named `myMovieCast`, with three elements and populate it with one comedian and two action heroes. Then iterate over each member of the array, and print out the string output of its act method call.

Now repeat the same for an array list of Actors `ArrayList<Actor>` named `myMovieCastList`.

Solution: In this problem, we are illustrating the concept of *polymorphism*, where objects of different types (`Comedian` and `ActionHero`) can have the same super type (`Actor` interface). A possible solution is shown below.

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

	// array of Actors

	Actor[] myMovieCast = new Actor[3];

	myMovieCast[0] = new Comedian();
	myMovieCast[1] = new ActionHero();
	myMovieCast[2] = new ActionHero();

	for(Actor myActor : myMovieCast) {
		System.out.println(myActor.act());
	}

	// repeat for an array list of Actors

	List<Actor> myMovieCastList = new ArrayList<Actor>();

	myMovieCastList.add(new Comedian());
	myMovieCastList.add(new ActionHero());
	myMovieCastList.add(new ActionHero());

	for(Actor actor : myMovieCastList) {
		System.out.println(actor.act());
	}
}
```