Programming in Java

Revision worksheet

for the help/revision session on the
9th May 2016

For each of the following questions you may use type in the code fragments if that assists with your answer but do remember that you won't have that facility during the examination.

1. Consider the following method `strange`:

```
1  package worksheet;
2
3  import java.util.ArrayList;
4  import java.util.Arrays;
5  import java.util.List;
6
7  public class Odd {
8    private static List<Integer> strange(List<Integer> list) {
9      List<Integer> res = new ArrayList<>();
10     list.stream().forEach(i -> res.add(i));
11
12     for (int i = list.size() - 1; i >= 0; i--) {
13       if (i % 2 == 0) {
14         res.add(list.get(i));
15       } else {
16         res.add(0, list.get(i));
17       }
18     }
19     return res;
20   }
21
22   private static <T> void sop(List<T> arg){
23     System.out.println(arg);
24   }
25
26   public static void main(String[] args) {
27     sop(strange(Arrays.asList(10, 20, 30)));
28     sop(strange(Arrays.asList(8, 2, 9, 7, 4)));
29     sop(strange(Arrays.asList()));
30     sop(strange(Arrays.asList(33, -1, 3, 17, -1, 11)));
31   }
32 }
```

What is the output produced when the `main` method is executed? What does the method `strange` compute in general?

Solution:

- `[20, 10, 20, 30, 30, 10]`

- `[2, 7, 8, 2, 9, 7, 4, 4, 9, 8]`
- `[]`
- `[-1, 17, 11, 33, -1, 3, 17, -1, 11, -1, 3, 33]`

Uses the index to decide where to place the new items. Odd position elements are added at the end; even position elements at the beginning. The list is traversed from back to front. This presumes that the index positions start at zero.

2. Write a method `removeBadPairs` that accepts an `ArrayList` of integers and removes any adjacent pair of integers in the list if the left element of the pair is larger than the right element of the pair. Every pair's left element is an even-numbered index in the list, and every pair's right element is an odd index in the list.

   For example, suppose a variable called `list` stores the following element values:

   ```
   [3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1]
   ```

   We can think of this list as a sequence of pairs:

   ```
   [3, 7] [9, 2] [5, 5] [8, 5] [6, 3] [4, 7] [3, 1]
   ```

   The pairs `9-2`, `8-5`, `6-3`, and `3-1` are "bad" because the left element is larger than the right one, so these pairs should be removed. So the call of `removeBadPairs(list);` would change the list to store

   ```
   [3, 7, 5, 5, 4, 7]
   ```

   If the list has an odd length, the last element is not part of a pair and is also considered "bad"; it should therefore be removed by your method.

   If an empty list is passed in, the list should still be empty at the end of the call. You may assume that the list passed is not `null`. You may not use any other arrays, lists, or other data structures to help you solve this problem, though you can create as many simple variables as you like.

Solution: Two possible solutions are shown:

```
package worksheet;

import java.util.ArrayList;

public class RemoveBadPairs {
  public static void removeBadPairs(ArrayList<Integer> list) {
    if (list.size() % 2 != 0) {
      list.remove(list.size() - 1);
    }
    for (int i = 0; i < list.size(); i += 2) {
      if (list.get(i) > list.get(i + 1)) {
        list.remove(i);
        list.remove(i);
        i -= 2;
      }
    }
  }

  public static void removeBadPairsAgain(ArrayList<Integer> list) {
    if (list.size() % 2 != 0) {
      list.remove(list.size() - 1);
    }
    for (int i = list.size() - 1; i > 0; i--) {
      if (i % 2 != 0 && list.get(i - 1) > list.get(i)) {
        list.remove(i);
        list.remove(i - 1);
      }
    }
```

```
  }
}
```

3. Write a method `rarestAge` that accepts as a parameter a map from students' names (strings) to their ages (integers), and returns the *least* frequently occurring age. Consider a map variable `m` containing the following key/value pairs:

```
{Alyssa=22, Char=25, Dan=25, Jeff=20, Casey=20, Kim=20,
 Morgan=25, Ryan=25, Stef=22}
```

Three people are age 20 (Jeff, Casey, and Kim), two people are age 22 (Alyssa and Stef), and four people are age 25 (Char, Dan, Morgan, and Ryan). So a call of `rarestAge(m)` returns `22` because only two people are that age.

If there is a tie (two or more rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair of `Kelly=22` to the map above, there would now be a tie of three people of age 20 (Jeff, Casey, Kim) and three people of age 22 (Alyssa, Kelly, Stef). So a call of `rarestAge(m)` would now return `20` because 20 is the smaller of the rarest values.

If the map passed to your method is `null` or empty, your method should throw an `IllegalArgumentException`. You may assume that no key or value stored in the map is `null`. Otherwise you should not make any assumptions about the number of key/value pairs in the map or the range of possible ages that could be in the map.

You may create one new set or map as auxiliary storage to solve this problem. You can have as many simple variables as you like. You should not modify the contents of the map passed to your method.

Solution: Four possible solutions are shown:

```
package worksheet;

import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class RarestAge {
  public static int rarestAge(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new TreeMap<>();
    for (String name : m.keySet()) {
      int age = m.get(name);
      if (counts.containsKey(age)) {
        counts.put(age, counts.get(age) + 1);
      } else {
        counts.put(age, 1);
      }
    }

    int minCount = m.size() + 1;
    int rareAge = -1;
```

```
    for (int age : counts.keySet()) {
      int count = counts.get(age);
      if (count < minCount) {
        minCount = count;
        rareAge = age;
      }
    }
    return rareAge;
  }

  public static int rarestAgeTwo(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new TreeMap<>();
    for (int age : m.values()) {
      if (!counts.containsKey(age)) {
        counts.put(age, 0);
      }
      counts.put(age, counts.get(age) + 1);
    }

    int rareAge = -1;
    for (int age : counts.keySet()) {
      if (rareAge < 0 || counts.get(age) < counts.get(rareAge)) {
        rareAge = age;
      }
    }
    return rareAge;
  }

  public static int rarestAgeThree(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new TreeMap<>();
    for (String name : m.keySet()) {
      if (counts.containsKey(m.get(name))) {
        counts.put(m.get(name), counts.get(m.get(name)) + 1);
      } else {
        counts.put(m.get(name), 1);
      }
    }

    int minCount = Integer.MAX_VALUE;
    // really big number to be overwritten

    for (int age : counts.keySet()) {
      minCount = Math.min(minCount, counts.get(age));
    }
    for (int age : counts.keySet()) {
      if (counts.get(age) == minCount) {
        return age;
      }
```

```
    }
    return -1;    // won't reach here
  }

  public static int rarestAgeFour(Map<String, Integer> m) {
    if (m == null || m.isEmpty()) {
      throw new IllegalArgumentException();
    }

    Map<Integer, Integer> counts = new HashMap<>();
    for (String name : m.keySet()) {
      if (!counts.containsKey(m.get(name))) {
        counts.put(m.get(name), 0);
      }
      counts.put(m.get(name), counts.get(m.get(name)) + 1);
    }

    int minCount = Integer.MAX_VALUE;
    // really big number to be overwritten

    for (int age : counts.keySet()) {
      minCount = Math.min(minCount, counts.get(age));
    }
    int rareAge = -1;
    for (int age : counts.keySet()) {
      if (counts.get(age) == minCount &&
          (rareAge < 0 || age < rareAge)) {
        rareAge = age;
      }
    }
    return rareAge;
  }
}
```

4. Consider the following method `something`:

```
1  package worksheet;
2
3  public class Something {
4    public static void something(int n) {
5      if (n < 0) {
6        System.out.print("-");
7        something(-n);
8      } else if (n < 10) {
9        System.out.println(n);
10     } else {
11       int two = n % 100;
12       System.out.print(two / 10);
13       System.out.print(two % 10);
14       something(n / 100);
15     }
16   }
17
18   public static void main(String[] args) {
19     something(7);
20     something(825);
21     something(38947);
22     something(612305);
23     something(-12345678);
24   }
25 }
```

What values are returned when the `main` method is executed? What does the method `something` compute in general?

Solution:

- `7`
- `258`
- `47893`
- `0523610`
- `-785634120`

5. Write a recursive method `isReverse` that accepts two strings as a parameter and returns `true` if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalisation), and `false` otherwise.

   For example, the string `"hello"` backwards is `"olleh"`, so a call of

   ```
   isReverse("hello", "olleh")
   ```

   would return `true`. Since the method is case-insensitive, you would also get a `true` result from a call of

   ```
   isReverse("Hello", "oLLEh")
   ```

The empty string, as well as any one-letter string, is considered to be its own reverse. The string could contain characters other than letters, such as numbers, spaces, or other punctuation; you should treat these like any other character.

The key aspect is that the first string has the same sequence of characters as the second string, but in the opposite order, ignoring case.

You may assume that the strings passed are not `null`. You are not allowed to construct any structured objects other than `String`s (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion. If you like, you may declare other methods to help you solve this problem, subject to the previous rules.

Solution: Four possible solutions are shown:

```
package worksheet;

public class IsReverse {
  public static boolean isReverse(String s1, String s2) {
    if (s1.length() == 0 && s2.length() == 0) {
      return true;
    } else if (s1.length() == 0 || s2.length() == 0) {
      return false;  // not same length
    } else {
      String s1first = s1.substring(0, 1);
      String s2last = s2.substring(s2.length() - 1);
      return s1first.equalsIgnoreCase(s2last) &&
             isReverse(s1.substring(1),
               s2.substring(0, s2.length() - 1));
    }
  }

  public static boolean isReverseTwo(String s1, String s2) {
    if (s1.length() != s2.length()) {
      return false;  // not same length
    } else if (s1.length() == 0 && s2.length() == 0) {
      return true;
    } else {
      s1 = s1.toLowerCase();
      s2 = s2.toLowerCase();
      return s1.charAt(0) == s2.charAt(s2.length() - 1) &&
             isReverseTwo(s1.substring(1, s1.length()),
               s2.substring(0, s2.length() - 1));
    }
  }

  public static boolean isReverseThree(String s1, String s2) {
    if (s1.length() == s2.length()) {
      return isReverse(s1.toLowerCase(), 0,
        s2.toLowerCase(), s2.length() - 1);
    } else {
      return false;   // not same length
    }
  }

  private static boolean isReverse(String s1, int i1,
                                   String s2, int i2) {
```

```
    if (i1 >= s1.length() && i2 < 0) {
      return true;
    } else {
      return s1.charAt(i1) == s2.charAt(i2) &&
               isReverse(s1, i1 + 1, s2, i2 - 1);
    }
  }


  public static boolean isReverseFour(String s1, String s2) {
    return reverse(s1.toLowerCase()).equals(s2.toLowerCase());
  }

  private static String reverse(String s) {
    if (s.length() == 0) {
      return s;
    } else {
      return reverse(s.substring(1)) + s.charAt(0);
    }
  }
}
```

6. You have been asked to extend a pre-existing class `Point` that represents 2-D (x, y) coordinates. The `Point` class includes the following constructors and methods:

Constructor/Method	Description
`public Point()`	constructs the point $(0, 0)$
`public Point(int x, int y)`	constructs a point with the given x/y coordinates
`public void setLocation(int x, int y)`	sets the coordinates to the given values
`public int getX()`	returns the x-coordinate
`public int getY()`	returns the y-coordinate
`public String toString()`	returns a String in standard "(x, y)" notation
`public double distanceFromOrigin()`	returns the distance from the origin $(0, 0)$, computed as the square root of $(x^2 + y^2)$

You are to define a new class called `Point3D` that extends this class through inheritance. It should behave like a `Point` except that it should be a 3-dimensional point that keeps track of a z-coordinate. You should provide the same methods as the superclass, as well as the following new behaviour.

Constructor/Method	Description
`public Point3D()`	constructs the point $(0, 0, 0)$
`public Point3D(int x, int y, int z)`	constructs a point with given $x/y/z$ coordinates
`public void setLocation(int x, int y, int z)`	sets coordinates to the given values
`public int getZ()`	returns the z-coordinate

Some of the existing behaviours from `Point` should behave differently on `Point3D` objects:

- When the original 2-parameter version of the `setLocation` is called, the 3-D point's x/y coordinates should be set as specified, and the z-coordinate should be set to 0.
- When a 3-D point is printed with `toString`, it should be returned in an `"(x, y, z)"` format that shows all three coordinates.
- A 3-D point's distance from the origin is computed using all three coordinates; it is equal to the square root of $(x^2 + y^2 + z^2)$.

You must also make `Point3D` objects comparable to each other using the `Comparable` interface. 3-D points are compared by x-coordinate, then by y-coordinate, then by z-coordinate.

- A `Point3D` object with a smaller x-coordinate is considered to be *less than* one with a larger x-coordinate.
- If two `Point3D` objects have the same x-coordinate, the one with the lower y-coordinate is considered *less*.
- If they have the same x and y-coordinates, the one with the lower z-coordinate is considered *less*.
- If the two points have the same x, y, and z-coordinates, they are considered to be *equal*.

Solution:

```
package worksheet;

public class Point3D extends Point implements Comparable<Point3D> {
  private int z;

  public Point3D() {
    this(0, 0, 0);
  }

  public Point3D(int x, int y, int z) {
    setLocation(x, y, z);
  }

  public int getZ() {
    return z;
  }

  public String toString() {
    return "(" + getX() + ", " + getY() + ", " + z + ")";
  }

  public void setLocation(int x, int y) {
    setLocation(x, y, 0);
  }

  public void setLocation(int x, int y, int z) {
    super.setLocation(x, y);
    this.z = z;
  }

  public double distanceFromOrigin() {
    return Math.sqrt(getX() * getX() + getY() * getY() + z * z);
  }

  public int compareTo(Point3D other) {
    if (getX() != other.getX()) {
      return getX() - other.getX();
    } else if (getY() != other.getY()) {
      return getY() - other.getY();
    } else {
      return z - other.z;
    }
  }
}
```

7. With reference to *exception handling* what is wrong with the following code fragments? You should discuss the concept of *checked* and *unchecked* exceptions in your answers where it is appropriate.

   (a)
   ```
   package worksheet.exception;
   import java.io.IOException;

   public class A {
     public static void start() throws IOException, RuntimeException {
   ```

```
    throw new RuntimeException("Not able to Start");
  }

  public static void main(String args[]) {
    try {
      start();
    } catch (Exception ex) {
      ex.printStackTrace();
    } catch (RuntimeException re) {
      re.printStackTrace();
    }
  }
}
```

Solution: This code will produce a compiler error on the line where the `RuntimeException` variable `re` is written in the `catch` block. since `Exception` is the super class of `RuntimeException`. Any `RuntimeException`s thrown by the `start()` method will be captured by the first `catch` block and execution will never reach the second `catch` block and that's the reason the compiler will flag an error

`exception java.lang.RuntimeException has already been caught".`

or something similar.

(b)
```
package worksheet.exception;
import java.io.IOException;

public class B {
  public class SuperClass {
    public void start() throws IOException {
      throw new IOException("Not able to open file");
    }
  }

  public class SubClass extends SuperClass {
    public void start() throws Exception {
      throw new Exception("Not able to start");
    }
  }
}
```

Solution: For this code the compiler will complain on sub class where the `start()` method gets overridden. As per rules of method overriding in Java, an overridden method can not throw *Checked Exceptions* which are higher in the exception hierarchy than the original method. Since here `start()` throws `IOException` in its super class, `start()` in the sub class can only throw either `IOException` or any sub class of `IOException` but not a super class of `IOException`, e.g., `Exception`.

(c)
```
package worksheet.exception;
import java.io.IOException;

public class C {
  public static void start() {
    System.out.println("Java Exception test");
```

```
    }

    public static void main(String args[]) {
      try {
        start();
      } catch (IOException ioe) {
        ioe.printStackTrace();
      }
    }
  }
```

Solution: In this Java Exception example the compiler will complain about the line where we handle `IOException`, since `IOException` is a *checked exception* and the `start()` method doesn't throw `IOException`, so the compiler will flag a compile error as

```
"exception java.io.IOException is never thrown in the
    body of the corresponding try statement"
```

If you change `IOException` to `Exception` the compiler error will disappear because `Exception` can be used to catch all `RuntimeExceptions` which don't require declaration in a `throws` clause.

8. Consider the following classes:

```
package worksheet;

public class Main {
  public static void main(String[] args) {
    Box var1 = new Box();
    Pill var2 = new Jar();
    Box var3 = new Cup();
    Box var4 = new Jar();
    Object var5 = new Box();
    Pill var6 = new Pill();
  }
}

class Cup extends Box {
  public void method1() {
    System.out.println("Cup 1");
  }

  @Override
  public void method2() {
    System.out.println("Cup 2");
    super.method2();
  }
}

class Pill {
  public void method2() {
    System.out.println("Pill 2");
  }
}
```

```
class Jar extends Box {
  public void method1() {
    System.out.println("Jar 1");
  }

  @Override
  public void method2() {
    System.out.println("Jar 2");
  }
}

class Box extends Pill {
  @Override
  public void method2() {
    System.out.println("Box 2");
  }

  public void method3() {
    method2();
    System.out.println("Box 3");
  }
}
```

What is the output of the following statements?

(a) `var1.method2();`
(b) `var2.method2();`
(c) `var3.method2();`
(d) `var4.method2();`
(e) `var5.method2();`
(f) `var6.method2();`
(g) `var1.method3();`
(h) `var2.method3();`
(i) `var3.method3();`
(j) `var4.method3();`
(k) `((Cup) var1).method1();`
(l) `((Jar) var2).method1();`
(m) `((Cup) var3).method1();`
(n) `((Cup) var4).method1();`
(o) `((Jar) var4).method2();`
(p) `((Box) var5).method2();`
(q) `((Pill) var5).method3();`
(r) `((Jar) var2).method3();`
(s) `((Cup) var3).method3();`
(t) `((Cup) var5).method3();`

If the statement produces more than one line of output, indicate the line breaks with slashes as in `"a / b / c"` to indicate three lines of output with `"a"` followed by `"b"` followed by `"c"`.

If the statement causes an error, answer with the phrase `"error"` to indicate this.

Solution:

(a) `Box 2`

(b) `Jar 2`

(c) `Cup 2/Box 2`

(d) `Jar 2`

(e) `error`

(f) `Pill 2`

(g) `Box 2/Box 3`

(h) `error`

(i) `Cup 2/Box 2/Box 3`

(j) `Jar 2/Box 3`

(k) `error`

(l) `Jar 1`

(m) `Cup 1`

(n) `error`

(o) `Jar 2`

(p) `Box 2`

(q) `error`

(r) `Jar 2/Box 3`

(s) `Cup 2/Box 2/Box 3`

(t) `error`

9. Given the following method `m`:

```
1  package worksheet;
2
3  import java.util.*;
4
5  public class M {
6    public static void main(String[] args) {
7
8      m(toMap(Arrays.asList("sheep", "wool", "house", "brick",
9        "cast", "plaster", "wool", "wool")));
10     m(toMap(Arrays.asList("munchkin", "blue", "winkie", "yellow",
11       "corn", "yellow", "grass", "green", "emerald", "green")));
12     m(toMap(Arrays.asList("pumpkin", "peach", "corn", "apple",
13       "apple", "apple", "pie", "fruit", "peach", "peach")));
14     m(toMap(Arrays.asList("lab", "MAL", "lion", "cat", "terrier",
15       "dog", "cat", "cat", "platypus", "animal", "nyan", "cat")));
16   }
17
18   public static void m(Map<String, String> m) {
19     Set<String> s = new TreeSet<>();
20     for (String key : m.keySet()) {
21       if (!m.get(key).equals(key)) {
22         s.add(m.get(key));
23       } else {
24         s.remove(m.get(key));
25       }
26     }
27     System.out.println(s);
28   }
29
30   private static Map<String, String> toMap(List<String> lst) {
31     Map<String, String> map = new HashMap<>();
32     for (int x = 0; x < lst.size(); x = x + 2) {
33       map.put(lst.get(x), lst.get(x + 1));
34     }
35     return map;
```

```
36      }
37  }
```

What output is printed for each of the maps and what does the method `m` compute in general?

Your answer should display the correct values in the correct order.

Solution:

```
[brick, plaster, wool]
[blue, green, yellow]
[apple, fruit, peach]
[MAL, animal, cat, dog]
```