CS 61B Fall 2021 — Test Review

Exam Prep Discussion 6: September 27, 2021

Note this worksheet is very long and is not expected to be finished in an hour.

1 Packages Have Arrived

In the following classes, cross out the lines that will result in an error (either during compilation or execution). Next to each crossed-out line write a replacement for the line that correctly carries out the evident intent of the erroneous line.

Each replacement must be a single statement. Change as few lines as possible.

After your corrections, what is printed from running **java P2.C5**?

```
1  package P1;                                          Write output here:
2  class C1 {
3      private int a = 1;                               ____________________________
4      protected int b = 2;
5      int c = 3;                                       ____________________________
6
7      public static int d() {                          ____________________________
8          return 13;
9      }
10     public void setA(int v) { a = v; }
11     public void setB(int v) { b = v; }
12     public void setC(int v) { c = v; }
13     public int getA() { return a; }
14     public int getB() { return b; }
15     public int getC() { return c; }
16
17     public String toString() {
18         return a + " " + getB() + " " + getC() + " " + d();
19     }
20 }
21 ____________________________________________
22
23 package P1;
24 class C2 extends C1 {
25     public C2() {}
26     public C2(int a, int b, int c) {
27         this.a = a;
28         this.b = b;
29         this.c = c;
30     }
31     public static int d() {
```

```
        return 14;
    }
    public C1 gen() {
        return new C3();
    }
}
_____________________________________________

package P1;
class C3 extends C2 {
    private int a = 15;
    public String toString() {
        return a + " " + getB() + " " + getC() + " " + d();
    }
}
_____________________________________________

package P2;
class C4 extends C2 {
    public int getB() {
        return 2 * b;
    }
    public C4(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public C4(int v) {
        this.a = this.b = this.c = v;
    }
}
_____________________________________________

package P2;
class C5 {
    public static void main(String... args) {
        C1 x = new C1();
        C2 y = new C4(20, 30, 40);
        C3 z = y.gen();

        System.out.println(x);
        System.out.println((P1.C2) y);
        System.out.println(z);
    }
}
```

Solution:

```
package P1;

public class C1 {

    private int a = 1;
    protected int b = 2;
    int c = 3;

    public static int d() {
        return 13;
    }
    public void setA(int v) { a = v; }
    public void setB(int v) { b = v; }
    public void setC(int v) { c = v; }
    public int getA() { return a; }
    public int getB() { return b; }
    public int getC() { return c; }

    public String toString() {
        return a + " " + getB() + " " + getC() + " " + d();
    }
}
```

Write output here:

____ 1 2 3 13 _____

____ 20 60 40 13 _____

____ 15 2 3 14 _____

```
package P1;

public class C2 extends C1 {

    public C2() {}
    public C2(int a, int b, int c) {

        setA(a);

        this.b = b;
        this.c = c;
    }
    public static int d() {
        return 14;
    }
    public C1 gen() {
        return new C3();
    }
}
```

```
package P1;

public class C3 extends C2 {
```

```
    private int a = 15;
    public String toString() {
        return a + " " + getB() + " " + getC() + " " + d();
    }
}
```

```
package P2;
class C4 extends P1.C2 {
    public int getB() {
        return 2 * b;
    }
    public C4(int a, int b, int c) {
        setA(a);
        this.b = b;
        setC(c);
    }
    public C4(int v) {
        super(v,v,v);
    }
}
```

```
package P2;
class C5 {
    public static void main(String... args) {
        P1.C1 x = new P1.C1();
        P1.C2 y = new C4(20, 30, 40);
        P1.C3 z = (P1.C3) y.gen();
        System.out.println(x);
        System.out.println((P1.C2) y);
        System.out.println(z);
    }
}
```

Fixes:
The following lines need to be fixed:
Line 2: In order to access `C1` in another package `P2`, it needs to be a `public class`.
Line 23: Similar logic to line 2–in order to access `C2` in another package, it needs to be a `public class`.
Line 27: `a` is a private variable, so subclasses like `C2` cannot access it directly–instead, it must use the `setA` method.

Line 41: Similar to line 2 and 23.
Line 50: Since we have not imported `P1`, we must use preface the class we want to use by its full package name.
Line 55: Similar to Line 27, `a` is a private variable.
Line 57: A variable with no declared access modifier is package private, which means it can only be accessed within the same package. `C4` is in package `P2`, so it must use the public method `setC` instead of directly accessing `c`.
Line 60: Again, `C4` cannot directly access `a` or `c`, but it can call its parent's constructor with `super`, which achieves the desired effect.
Line 68, 69: Similar to Line 50; without importing `P1`, the full package name must be used.
Line 70: `y` has static type `C2`, and `C2.gen` has return type `C1`. However, we know that the method actually returns an object of type `C3`, so casting allows the assignment to compile.
Print output:
Line 72: `C1`'s `toString` method simply prints the `a, b, c`, and `d()` values.
Line 73: `C4` has `a, b, c` as `20, 30, 40` respectively. It inherits the `toString` method from `C1`. However, its `getB` method overrides the `getB` method in `C1`, so `getB()` will return 60. Note that the `d()` method is static, so the method that is run is decided at compile time (there is no overriding for static methods). Thus, at compile time, the compiler finds `getString()` inside of `C1` and also uses `C1`'s `d()` method. The final values printed are `20, 60, 40, 13`.
Line 74: `y.gen()` calls the no-argument `C3` constructor. Note that `C3` has its own `private a` with a value of 15. However, it still inherits `b` and `c` from `C1`, and its `d()` method from `C2`. This gives the final output `15 2 3 14`.

2 Iterator of Iterators

Implement an `IteratorOfIterators` which will accept as an argument a `List` of `Iterator` objects containing `Integers`. The first call to `next()` should return the first item from the first iterator in the list. The second call to `next()` should return the first item from the second iterator in the list. If the list contained `n` iterators, the `n+1`th time that we call `next()`, we would return the second item of the first iterator in the list.

Note that if an iterator is empty in this process, we continue to the next iterator. Then, once all the iterators are empty, `hasNext` should return **`false`**. For example, if we had 3 `Iterators` A, B, and C such that A contained the values `[1, 3, 4, 5]`, B was empty, and C contained the values `[2]`, calls to `next()` for our `IteratorOfIterators` would return `[1, 2, 3, 4, 5]`.

```
1  import java.util.*;
2  public class IteratorOfIterators  ______________________________ {
3
4
5      public IteratorOfIterators(List<Iterator<Integer>> a) {
6
7
```

```
8
9
10
11
12
13      }
14
15      @Override
16      public boolean hasNext() {
17
18
19
20
21      }
22
23
24
25      @Override
26      public Integer next() {
27
28
29
30
31      }
32  }
```

Solution: Here is a video walkthrough of the solution.

```
1   public class IteratorOfIterators implements Iterator<Integer> {
2       LinkedList<Iterator<Integer>> iterators;
3
4       public IteratorOfIterators(List<Iterator<Integer>> a) {
5           iterators = new LinkedList<>();
6           for (Iterator<Integer> iterator : a) {
7               if (iterator.hasNext()) {
8                   iterators.add(iterator);
9               }
10          }
11      }
12
13      @Override
14      public boolean hasNext() {
15          return !iterators.isEmpty();
16      }
17
18      @Override
19      public Integer next() {
```

```
20          if (!hasNext()) {
21              throw new NoSuchElementException();
22          }
23          Iterator<Integer> iterator = iterators.removeFirst();
24          int ans = iterator.next();
25          if (iterator.hasNext()) {
26              iterators.addLast(iterator);
27          }
28          return ans;
29      }
30  }
```

Explanation: In the constructor, we make sure the iterator is not empty and add it to our list of possible iterators. For `hasNext`, we make sure that there is an iterator for us to use.

For `next`, we first make sure that there is a possible next element. If so, we get the next element from the current iterator by removing the front of our list. If the iterator still has elements left, we put it back on the end of the list for future iterations.

Alternate Solution: Although this solution provides the right functionality, it is not as efficient as the first one.

```
public class IteratorOfIterators implements Iterator<Integer> {
    LinkedList<Integer> l;

    public IteratorOfIterators(List<Iterator<Integer>> a) {
        l = new LinkedList<>();
        while (!a.isEmpty()) {
            Iterator<Integer> curr = a.remove(0);
            if (curr.hasNext()) {
                l.add(curr.next());
                a.add(curr);
            }
        }
    }

    @Override
    public boolean hasNext() {
        return !l.isEmpty();
    }

    @Override
    public Integer next() {
        if(!hasNext()) {
            throw new NoSuchElementException();
        }
        return l.removeFirst();
    }
}
```

Explanation: This solution is essentially the same as the first, except we preprocess all the elements from all iterators before going into `hasNext` or `next`. This is less efficient because we may not need all these elements; for example, what if there are a million elements but our iterator is only called twice?

3 DMS Comparator

Implement the Comparator `DMSComparator`, which compares `Animal` instances. An `Animal` instance is greater than another `Animal` instance if its **dynamic type** is more *specific*. See the examples to the right below.

In the second and third blanks in the `compare` method, **you may only use the integer variables predefined** (`first`, `second`, etc), **relational/equality operators** (`==`, `>`, etc), **boolean operators** (`&&` and `||`), **integers**, and **parentheses**.

As a *challenge*, use equality operators (`==` or `!=`) and no relational operators (`>`, `<=`, etc). There may be more than one solution.

```
class Animal {
   int speak(Dog a) { return 1; }
   int speak(Animal a) { return 2; }
}
class Dog extends Animal {
   int speak(Animal a) { return 3; }
}
class Poodle extends Dog {
   int speak(Dog a) { return 4; }
}
```

Examples:

```
Animal animal = new Animal();
Animal dog = new Dog();
Animal poodle = new Poodle();

compare(animal, dog) // negative number
compare(dog, dog) // zero
compare(poodle, dog) // positive number
```

```
1  public class DMSComparator implements ____________________________ {
2
3      @Override
4      public int compare(Animal o1, Animal o2) {
5          int first = o1.speak(new Animal());
6          int second = o2.speak(new Animal());
7          int third = o1.speak(new Dog());
8          int fourth = o2.speak(new Dog());
9
10         if (_____________________________________________________) {
11             return 0;
12
13         } else if (_____________________________________________) {
14             return 1;
15         } else {
16             return -1;
17         }
18     }
19 }
```

Solution: Here is a video walkthrough of the solution.

```
public class DMSComparator implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());

        if (first == second && third == fourth) {
            return 0;
        } else if (first > second || third > fourth) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

Explanation:

We know that we should return 0 when the dynamic types of `o1` and `o2` are the same. However, just checking `first == second` is insufficient. Consider the case where you have `o1` with dynamic type `Dog` and `o2` with dynamic type `Poodle`. During compilation, both of these will choose the method `speak(Animal a)`. During runtime, `first` will be `3`, since `Dog.speak(Animal a)` overrides `Animal.speak(Animal a)`. `second` will also be 3: `Poodle` does not have a `speak(Animal)` method, so it goes to its superclass `Dog` and finds `Dog.speak(Animal a)`. Thus, we must also check `third == fourth` in the first case.

For the case of returning 1, note that if `o1` is a `Poodle` while `o2` is not, we should return 1. In this case, `fourth = o2.speak(Dog)` will return 4, while `o1.speak(Dog)` will return 1. Thus, we check if `fourth > third`; if it is, `o1` is more specific than `o2`. Then, we consider the case of `o1` being a `Dog` and `o2` being an `Animal`. In this case, `o1.speak(Animal)` will return `3` (since at runtime tye dynamic type `Dog` also has a `speak(Animal)` method) whereas `o2.speak(Animal)` will return 2. This gives us the other condition, `first > second`.

Challenge Solution:

```
public class DMSComparator implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        int first = o1.speak(new Animal());
        int second = o2.speak(new Animal());
        int third = o1.speak(new Dog());
        int fourth = o2.speak(new Dog());
```

```
9
10          if (first == second && third == fourth) {
11              return 0;
12          } else if (third == 4 || (first == 3 && second == 2)) {
13              return 1;
14          } else {
15              return -1;
16          }
17      }
18  }
```

Explanation:

The first if statement is the same as the solution above.

If we reach the second case and `o1` is a `Poodle`, we know `o2` must be a `Dog` or an `Animal` (or we would have returned 0 in the first case). Thus, we can immediately return 1 if `o1` is a `Poodle`. To check if `o1` is a `Poodle`, we can simply check if `third == 4` (since only Poodles can return `4`).

There is one other case in which we should return 1: when `o1` is a `Dog` and `o2` is an `Animal`. If `o1` is a `Dog`, `o2.speak(Animal)` should return 3, so we check if `first == 3`. To check if `o2` is an `Animal`, we ensure that `o2.speak(Animal)` returns `2` (if it had dynamic type `Dog` or `Poodle`, it would use the method in `Dog` which returns 3). Thus, we add the condition `second == 2`.

4 Partition

Implement `partition`, which takes in an `IntList lst` and an integer `k`, and *destructively* partitions `lst` into `k IntLists` such that each list has the following properties:

1. It is the **same** length as the other lists. If this is not possible, i.e. `lst` cannot be equally partitioned, then the later lists should be **one** element smaller. For example, partitioning an `IntList` of length 25 with `k = 3` would result in partitioned lists of lengths 9, 8, and 8.
2. Its ordering is consistent with the ordering of `lst`, i.e. items in earlier in `lst` must **precede** items that are later.

These lists should be put in an array of length `k`, and this array should be returned. For instance, if `lst` contains the elements 5, 4, 3, 2, 1, and `k = 2`, then a **possible** partition (note that there are many possible partitions), is putting elements 5, 3, 2 at index 0, and elements 4, 1 at index 1.

You may assume you have the access to the method `reverse`, which destructively reverses the ordering of a given `IntList` and returns a pointer to the reversed `IntList`. You may not create any `IntList` instances. You may not need all the lines.

Hint: You may find the % operator helpful.

```
1  public static IntList[] partition(IntList lst, int k) {
2      IntList[] array = new IntList[k];
3      int index = 0;
4      IntList L = ____________________________________
5      while (L != null) {
6
7          ___________________________________________
8
9          ___________________________________________
10
11         ___________________________________________
12
13         ___________________________________________
14
15         ___________________________________________
16
17         ___________________________________________
18
19         ___________________________________________
20     }
21     return array;
22 }
```

Solution: Here is a video walkthrough of the solution.

```
1  public static IntList[] partition(IntList lst, int k) {
2      IntList[] array = new IntList[k];
3      int index = 0;
4      IntList L = reverse(lst);
5      while (L != null) {
6          IntList prevAtIndex = array[index];
7          IntList next = L.rest;
8          array[index] = L;
9          array[index].rest = prevAtIndex;
10         L = next;
11         index = (index + 1) % array.length;
12     }
13     return array;
14 }
```

Explanation: We reverse our `IntList` so that we can build up each element of the `IntList[]` array backwards–in general, it is much easier to build an `IntList` backward than forward.

The general idea is to initialize each element in the array to `null`, then put an element of `L` inside the correct index by assigning `array[index] = L`. Then, we get whatever we've built up so far (`prevAtIndex`) and add it to the end of our `rest` element so that we have the entire IntList again with one element at the front.

Afterwards, we advance `L` to the next element and increment the index.