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;
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() {
```

Write output here:

_____

_____

_____

```java
32          return 14;
33      }
34      public C1 gen() {
35          return new C3();
36      }
37  }
38  ———————————————————————————————————————————
39
40  package P1;
41  class C3 extends C2 {
42      private int a = 15;
43      public String toString() {
44          return a + " " + getB() + " " + getC() + " " + d();
45      }
46  }
47  ———————————————————————————————————————————
48
49  package P2;
50  class C4 extends C2 {
51      public int getB() {
52          return 2 * b;
53      }
54      public C4(int a, int b, int c) {
55          this.a = a;
56          this.b = b;
57          this.c = c;
58      }
59      public C4(int v) {
60          this.a = this.b = this.c = v;
61      }
62  }
63  ———————————————————————————————————————————
64
65  package P2;
66  class C5 {
67      public static void main(String... args) {
68          C1 x = new C1();
69          C2 y = new C4(20, 30, 40);
70          C3 z = y.gen();
71
72          System.out.println(x);
73          System.out.println((P1.C2) y);
74          System.out.println(z);
75      }
76  }
```

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+1th 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]`.

```java
import java.util.*;
public class IteratorOfIterators  _____ {



    public IteratorOfIterators(List<Iterator<Integer>> a) {








    }

    @Override
    public boolean hasNext() {





    }



    @Override
    public Integer next() {





    }
}
```

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 {                        Examples:
    int speak(Dog a) { return 1; }
    int speak(Animal a) { return 2; }   Animal animal = new Animal();
}                                       Animal dog = new Dog();
class Dog extends Animal {               Animal poodle = new Poodle();
    int speak(Animal a) { return 3; }
}                                       compare(animal, dog) // negative number
class Poodle extends Dog {              compare(dog, dog) // zero
    int speak(Dog a) { return 4; }     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 }
```

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  }
```