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?

```java
package P1;

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();
    }
}

package P1;

class C2 extends C1 {
    public C2() {}

    public C2(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public static int d() {
```
Test Review

```
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);
    }
```
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 {
    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

public class DMSComparator implements ____________ {
    @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 (__________________________________________) {
            return 0;
        }
        else if (______________________________________) {
            return 1;
        } else {
            return -1;
        }
    }
}
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.

```java
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 }
```