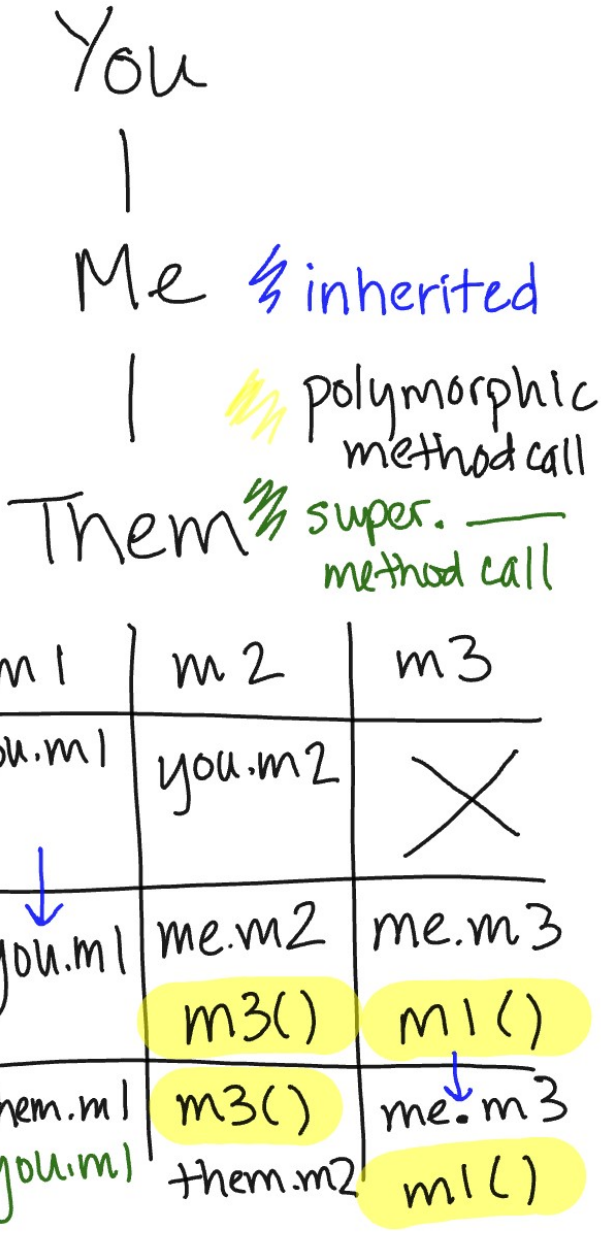
Inheritance Mystery

```
public class Me extends You {
   public void method2() {
      System.out.println("me.method2()");
      method3();
   }

   public void method3() {
      System.out.println("me.method3()");
      method1();
   }

}

public class Them extends Me {
   public void method1() {
      System.out.println("them.method1()");
      super.method1();
   }

   public void method2() {
      method3();
      System.out.println("them.method2()");
   }
}

public class You {
   public void method1() {
      System.out.println("you.method1()");
   }

   public void method2() {
      System.out.println("you.method2()");
   }
}
```




```
// Example of a typical "inheritance mystery" problem you might see
//  on a CSE142 final exam.
public class Client {
   public static void main(String[] args) {
      You[] us = { new You(), new Me(), new Them() };

      for (int i = 0; i < us.length; i++) {
         us[i].method1();
         System.out.println();
         us[i].method2();
         System.out.println();
         System.out.println("------");
      }
   }
}
```

Output:

```
you.method1()
you.method2()
------
you.method1()
me.method2()
me.method3()
you.method1()
------
them.method1()
you.method1()
me.method3()
them.method1()
you.method1()
them.method2()
```

ArrayList

Write a static method manyStrings that takes an ArrayList of Strings and an integer n as parameters and that replaces every String in the original list with n of that String. For example, suppose that an ArrayList called "list" contains the following values:

```
("squid", "octopus")
```

And you make the following call:

```
manyStrings(list, 2);
```

Then list should store the following values after the call:

```
("squid", "squid", "octopus", "octopus")
```

As another example, suppose that list contains the following:

```
("a", "a", "b", "c")
```

and you make the following call:

```
manyStrings(list, 3);
```

Then list should store the following values after the call:

```
("a", "a", "a", "a", "a", "a", "b", "b", "b", "c", "c", "c")
```

You may assume that the ArrayList you are passed contains only Strings and that the integer n is greater than 0.

```
public static void manyStrings(ArrayList<String> list, int n){

    // could also use
    // for (int i=0; i< list.size(); i += n){
    for (int i=list.size()-1; i>=0; i--){
        String target = list.get(i);
        for (int j=0; j<(n-1); j++){
            list.add(i, target);
        }
    }

}
```

already one copy in list, need to add n-1 more to have n total

Critters

Write a class called Salmon that extends the Critter class. The instances of the Salmon class should always infect when an enemy is in front of them and should otherwise hop if the space in front of them is empty. When they can't either infect or hop, they should randomly choose between turning left and turning right with each choice being equally likely. They should be colored pink (there is a color constant for pink). They should initially display themselves with the text "<*>" before they have made any turns. After they have made a turn, they should display themselves with the text "<L>" if their last turn was left and "<R>" if their last turn was right.

As in assignment 9, all fields must be declared private and fields that need to be initialized to a non-default value must be set in a constructor.

```
public class Salmon extends Critter {
    private String display;
    private Random r;

    public Salmon(){
        display = "*";
        r = new Random();
    }

    public Color getColor(){
        return Color.PINK;
    }

    public Action getMove(CritterInfo info){
        Neighbor front = info.getFront();
        if (front == Neighbor.OTHER){
            return Action.INFECT;
        } else if (front == Neighbor.EMPTY){
            return Action.HOP;
        } else { // make a turn
            int choose = r.nextInt(2);
            if (choose == 0){
                display = "<L>";
                return Action.LEFT;
            } else {
                display = "<R>";
                return Action.RIGHT;
            }
        }
    }

    public String toString(){
        return display;
    }
}
```

we need randomness in movement and storing it as a field means we only make one constructor in the whole class (needed because we need to initialize fields to non-default values)

update display field here because only in these places do we know the Salmon turned left or right

Line-Based File Processing

Write a static method called formatList that takes a Scanner connected to an input file as a parameter and prints to System.out the input with certain lines indented and with asterisks. The lines to be indented all begin with at least one period. These leading period(s) should not be printed. For each line with leading period(s), you should print the text on that line (not including the period(s)) preceded by four spaces of indentation per period, an asterisk, and a space.

For example, consider the following input file:

```
CSE schedule

   .week one
..static methods
..System.out.println()
..expressions

.week two
..for loops
..constants
..parameters
```

Then suppose the text above is stored in a Scanner called input and we make this call:

```
formatList(input);
```

The method should print the following output to System.out:

```
CSE schedule

    * week one
        * static methods
        * System.out.println()
        * expressions

    * week two
        * for loops
        * constants
        * parameters
```

Notice that input lines can be blank lines, and that input lines can contain periods of their own. For example, the periods in "System.out.println()" are not interpreted as indentation because they are not at the beginning of the line. Also note that lines without leading periods (like "CSE schedule") are printed as-is, with no indentation or asterisks.

You may not construct any extra data structures to solve this problem, though you may create as many String or primitive variables as you like.

```
public static void formatList(Scanner input){
    while(input.hasNextLine()){
        String line = input.nextLine();
        if (line.startsWith(".")){
            while (line.startsWith(".")){
                S.o.p("    ");
                line = line.substring(1);
            }
            S.o.pln("* " + line);
        }else{
            S.o.pln(line);
        }
    }
}
```

typical line-based processing set up

for each "." at beginning, indent and remove that "." so we can process the rest

Token-Based File Processing

Write a static method called switchData that takes as a parameter a Scanner containing a label followed by a sequence of integers and that prints to System.out the same information with each successive pair of integers switched in order. For example, suppose that a Scanner called data contains the following tokens:

```
Jan 1 2 3 4 5 6
```

Here the label is "Jan". The label will always be a single word that appears at the beginning. After the label, we have a series of six integers. If we make the following call:

```
switchData(data);
```

the method should produce the following output:

```
Jan 2 1 4 3 6 5
```

Notice that the first pair of integers (1, 2) has been switched (2, 1), and the second pair of integers (3, 4) has been switched (4, 3), and so on.

This first example involved sequential integers to make the switching more obvious, but this won't always be the case. You also shouldn't assume that you have an even number of integers. If there is an odd number of integers, then the final value should not be moved. For example, if the Scanner had instead contained these tokens:

```
Feb 38 14 79 4 -3
```

then the method would have produced the following output:

```
Feb 14 38 4 79 -3
```

There will always be a one-word label, but the list of integers might be empty, in which case the method simply prints the label on a line by itself. Your method should produce a complete line of output. In other words, if it is called n times, it will produce n lines of output. You may assume that the input is legal (a one-word label followed by 0 or more integer values). You may not construct any extra data structures to solve this problem.

```
public static void switchData(Scanner input){
    String label = input.next();
    S.o.p(label);

    while (input.hasNextInt()){
        int num1 = input.nextInt();
        if (input.hasNextInt()){
            int num2 = input.nextInt();
            S.o.p(" " + num2);
        }
        S.o.p(" " + num1);

    }
```

We have guarantee that the file will contain the label

in case there are an odd number of integers following the label

Arrays Programming (harder)

Write a static method named insertMiddle that accepts two arrays of integers *a* and *b* as parameters and returns a new array containing elements from the first half of *a* followed by all the elements of *b* followed by elements from the second half of *a*. For example, consider the following two arrays:

```
int[] a = {2, 4, 6, 8, 10};
int[] b = {1, 1, 1};
```

The call insertMiddle(a, b); should return the following array:

```
{2, 4, 1, 1, 1, 6, 8, 10}
```

Notice that if a has an odd length, its shorter half goes first.

You may not construct any extra data structures or String objects to solve this problem. You may not modify the arrays that are passed in.

```
public static int[] insertMiddle(int[] a, int[] b) {
    int[] result = new int[a.length + b.length];
    // first 1/2 of a
    for (int i=0; i<a.length/2; i++) {
        result[i] = a[i];
    }
    // all of b in middle
    for (int i=0; i<b.length; i++) {
        result[a.length/2 + i] = b[i];
    }
    // second 1/2 of a
    for (int i=a.length/2; i<a.length; i++) {
        result[b.length+i] = a[i];
    }
}
```

even handles odd-length cases!

starting at index a.length/2 (first unfilled), fill in everything side by side

i taking on values of indexes in a array with elements that still need to be copied over

how far we are into a, plus we know we already copied over the entirety of b