Developing the prelim string problem
/** s is a sequence of words with each pair of words separated
* by one or more blanks. Return a list of the Pig-Latin
* translations of the words, with no duplicates */
public static ArrayList<String> m(String s) {

A few points to be constantly aware of
• Focus on one thing at a time.
• Use abstraction.
• Keep things simple
• Avoid case analysis where possible
• Don’t introduce a variable unless you need it.

Word: a sequence of >= 1 lowercase letters

Word: a sequence of >= 1 lowercase letters

Use a loop to process string s
/** s is a sequence of words with each pair of words separated
* by one or more blanks. Return a list of the Pig-Latin
* translations of the words, with no duplicates */
public static ArrayList<String> m(String s) {

Outline the while-loop
s= s.trim();
// inv: All processed words have been removed from s,
// and s has no surrounding blanks
while ( s.length() > 0 ) {
    Process first word of s and remove it from s
}

Outline the while-loop
s= s.trim();
// inv: All processed words have been removed from s,
// and s has no surrounding blanks
while ( s.length() > 0 ) {
    // Process first word of s and remove it from s
    int k= s.indexOf( "  ");
    int k= s.length();
    if (k < 0)  k= s.length();
    s= s.substring(k).trim();
}

Problem: the last word has no blank after it!
Whenever you write
b[k] or s.charAt[k] or s.substring(k, k) or list.get(k), etc.
ask yourself whether index k is in bounds.

Which kind of loop?
for (int k= 0; k < s.length(); k= k+1) {}
Stepwise refinement

Now we can work on processing a word, which has to do with constructing the ArrayList and adding the Pig Latin of non-duplicate words. 

Stepwise refinement: Take one (small) step at a time. Focus on the most important one at the moment.

Examples of steps:
- Implement an English statement by a sequence of statements
- Decide on using a loop
- Stub in a new method (Specification and header, with empty body) because of duplicate code or to remove complexity
- Add a local variable or field
- Replace an English statement by an equivalent Java statement

Listening to events: mouse click, mouse movement into or out of a window, a keystroke, etc.

- An event is a mouse click, a mouse movement into or out of a window, a keystroke, etc.
- To be able to "listen to" a kind of event, you have to:
  1. Have some class C implement an interface IN that is connected with the event.
  2. In class C, override methods required by interface IN; these methods are generally called when the event happens.
  3. Register an object of class C as a listener for the event. That object’s methods will be called when event happens.

We show you how to do this for clicks on buttons, clicks on components, and keystrokes.

Listening to a JButton

1. Implement interface ActionListener:
   ```java
   public class C extends JFrame implements ActionListener {
     ...
     
     public void actionPerformed(ActionEvent e) {
     ...
   }
   ```

2. In class C override actionPerformed, which is to be called when button is clicked:
   ```java
   /** Process click of button */
   public void actionPerformed(ActionEvent e) {
     ...
   }
   ```

3. Add an instance of class C as an "action listener" for button:
   ```java
   button.addActionListener(this);
   ```

What is a JButton?

Instance: associated with a “button” on the GUI, which can be clicked to do something

```
jb1= new JButton(); // jb1 has no text on it
jb2= new JButton("first"); // jb2 has label "first" on it
jb2.isEnabled(); // true iff a click on button can be detected
jb2.setEnabled(b); // Set enabled property
jb2.addActionListener(object); // object must have a method, which is called when button jb2 clicked (next page)
```

At least 100 more methods; these are most important

JButton is in package javax.swing
class ButtonDemo1 extends JFrame {
    /** Class inv: exactly one of eastB, westB is enabled */
    JButton westB = new JButton("west");
    JButton eastB = new JButton("east");
    public ButtonDemo1(String t) {
        super(t);
        Container cp = getContentPane();
        cp.add(westB, BLayout.WEST);
        cp.add(eastB, BLayout.EAST);
        westB.setEnabled(false);
        eastB.setEnabled(true);
        pack(); setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        boolean b = eastB.isEnabled();
        eastB.setEnabled(!b);
        westB.setEnabled(b);
    }
}

/* listen to a Button */
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}

Class Square

/** Complement the "has pink disk" property */
public void complementDisk() {
    hasDisk = ! hasDisk;
    repaint(); // Ask the system to repaint the square
}

/** A JPanel that is painted */
A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.

A JPanel that is painted

A JPanel that is painted

- The JFrame content pane has a JPanel in its CENTER and a "reset" button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears. Click the rest button and all pink circles disappear.

These are different kinds of events, and they need different listener methods.
**A class that listens to a mouse click in a Square**

```java
import java.awt.event.*;
import javax.swing.*;

public class MouseDemo2 extends JFrame {
    public MouseDemo2() {
        jb = new JButton("reset");
        wButt = new JButton("blue");
        eButt = new JButton("red");
        ... 
    }
    ... 
}
```

**Listening to the keyboard**

```java
public class AllCaps extends KeyAdapter {
    ... 
    public void keyPressed(KeyEvent e) {
        char typedChar = e.getKeyChar();
        capsLabel.setText("'" + typedChar + "'").toUpperCase();
    }
    ... 
}
```

**MouseDemo2**

```java
public class MouseDemo2() {
    ... 
    jb.addActionListener(this);
    wButt.addActionListener(this);
    eButt.addActionListener(this);
    ... 
}
```

**In package java.swing.event**

```java
public interface MouseListener {
    public void mouseClicked(MouseEvent e);
    public void mouseEntered(MouseEvent e);
    public void mouseExited(MouseEvent e);
    public void mousePressed(MouseEvent e);
    public void mouseReleased(MouseEvent e);
}
```

**Our class overrides only the method that processes mouse clicks**

```java
public class AllCaps extends KeyAdapter {
    ... 
    public void keyPressed(KeyEvent e) {
        char typedChar = e.getKeyChar();
        capsLabel.setText("'" + typedChar + "'").toUpperCase();
    }
    ... 
}
```

**MD2**

```java
public class MD2 extends JFrame {
    public MD2() {
        ... 
        MouseEvents me = new MouseEvents();
        JFrame jb = new JFrame("reset");
        jb.addMouseListener(me);
        jb.addMouseMovementListener(me);
        jb.addMouseWheelListener(me);
    }
    public void mouseClicked(MouseEvent e) {
        ... 
    }
    public void mouseEntered(MouseEvent e) {
        ... 
    }
    public void mouseExited(MouseEvent e) {
        ... 
    }
    public void mousePressed(MouseEvent e) {
        ... 
    }
    public void mouseReleased(MouseEvent e) {
        ... 
    }
}
```
Problem: can’t give a function as a parameter:

```java
public void m() { …
eButt.addActionListener(aP);
}
```

<table>
<thead>
<tr>
<th>Why not just give eButt the function to call? Can’t do it in Java! Can in some other languages</th>
</tr>
</thead>
</table>

```java
public void m() { …
eButt.addActionListener(new C());
}
```

<table>
<thead>
<tr>
<th>Java says: provide class C that wraps method; give eButt an object of class C</th>
</tr>
</thead>
</table>

```java
public class C implements IN {
    public void aP(ActionEvent e) { body }
}
```

<table>
<thead>
<tr>
<th>C must implement interface IN that has abstract method aP</th>
</tr>
</thead>
</table>

Solution to problem: Make BeListener an inner class.

```java
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt; …;
    public ButtonDemo3() { … }
    public void actionPerformed(ActionEvent e) { … }
    private class BeListener implements ActionListener { … }
}
```

<table>
<thead>
<tr>
<th>Inside-out rule then gives access to wButt, eButt</th>
</tr>
</thead>
</table>

We demo this using ButtonDemo3

Have a class for which only one object is created?

Use an anonymous class.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

```java
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt; …;
    public ButtonDemo3() { … }
    public void actionPerformed(ActionEvent e) { … }
    private class BeListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { body }
    }
}
```

<table>
<thead>
<tr>
<th>1 object of BeListener created. Ripe for making anonymous</th>
</tr>
</thead>
</table>

Making class anonymous will replace `new BeListener()`

<table>
<thead>
<tr>
<th>Expression that creates object of BeListener</th>
</tr>
</thead>
</table>

```java
private class BeListener implements ActionListener {
    declarations in class
} ...
```

1. Write `new`
2. Write `ActionListener`
3. Put in arguments of constructor call
4. Write `ActionListener()`
5. Put in class body

Replace `new BeListener()` by new-expression

```
BDDemo3
wButt … eButt … BD3
aPerf(… eButt … wButt …)
listens to wButt

BeLis
aPerf(… eButt … wButt …)
listens to eButt but can’t reference fields

BeLis@80
```

Make BeListener an inner class.

Inside-out rule says that methods in here can reference all the fields and methods.

```java
private class BeListener implements ActionListener {
    public void actionPerformed(ActionEvent e) { body }
}
```

Just as you can declare variables and methods within a class, you can declare a class within a class.

```java
class BeListener implements ActionListener {
    public void actionPerformed(ActionEvent e) { body }
}
```

Have a class for which only one object is created?

Use an anonymous class.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

```java
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt; …;
    public ButtonDemo3() { … }
    public void actionPerformed(ActionEvent e) { … }
    private class BeListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { body }
    }
}
```

<table>
<thead>
<tr>
<th>Inside-out rule then gives access to wButt, eButt</th>
</tr>
</thead>
</table>

We demo this using ButtonDemo3

Have a class for which only one object is created?

Use an anonymous class.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

```java
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt; …;
    public ButtonDemo3() { … }
    public void actionPerformed(ActionEvent e) { … }
    private class BeListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { body }
    }
}
```

<table>
<thead>
<tr>
<th>Inside-out rule then gives access to wButt, eButt</th>
</tr>
</thead>
</table>

We demo this using ButtonDemo3

Have a class for which only one object is created?

Use an anonymous class.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

```java
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt; …;
    public ButtonDemo3() { … }
    public void actionPerformed(ActionEvent e) { … }
    private class BeListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { body }
    }
}
```

<table>
<thead>
<tr>
<th>Inside-out rule then gives access to wButt, eButt</th>
</tr>
</thead>
</table>

We demo this using ButtonDemo3

Have a class for which only one object is created?

Use an anonymous class.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

```java
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt; …;
    public ButtonDemo3() { … }
    public void actionPerformed(ActionEvent e) { … }
    private class BeListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { body }
    }
}
```

<table>
<thead>
<tr>
<th>Inside-out rule then gives access to wButt, eButt</th>
</tr>
</thead>
</table>

We demo this using ButtonDemo3

Have a class for which only one object is created?

Use an anonymous class.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.