Focus Management

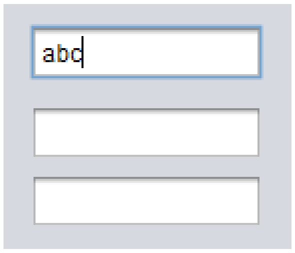
We will touch only briefly on focus management. There is much more available on Oracle's Java website. Focus refers to the component that is currently active. In particular, the listeners are only activated for the component that currently has focus. This can be a problem sometimes when you intend to have a component activated (e.g. a textfield for typing letters) but something else has focus instead, so nothing happens when you type keys.

You can traverse the focusable components of an application by hitting the tab key, or in reverse with shift-tab, or by click on them with the mouse.

The setFocusable(true) method will add a component to the traversable list of focusable components.

To programmatically set a component to have focus, use the requestFocusInWindow() method.

Given the following GUI, if we wanted to be able to move to the next JTextField upon hitting enter then we could write the following event handlers:



```
private void jTextField1KeyPressed(KeyEvent evt) {
    if (evt.getKeyCode()==10)
        jTextField2.requestFocusInWindow();
}

private void jTextField2KeyPressed(KeyEvent evt) {
    if (evt.getKeyCode()==10)
        jTextField3.requestFocusInWindow();
}
```

Etc.

It is possible to set the focus traversal policy, determine which component has focus (JFrame.getFocusOwner() returns a reference to the object with focus), change the keys to traverse focus, perform input validation before shifting focus, etc.

One question that has come up is how to catch all key presses. When a form has multiple components in it, we might want to be able to get any keypress. For example, if we have a game where WASD should control the player but there are items that could get focus in the JFrame or JPanel then how can we tell which key is pressed without adding keyevents to every component?

To do this we can override methods in the KeyboardFocusManager. The KeyBoardFocusManager is used to manage keyboard events. There is a central method that dispatches each event to the component with focus and we can put our own code in here to catch any key.

The following can be added to the JFrame constructor:

```
KeyboardFocusManager.getCurrentKeyboardFocusManager()
  .addKeyEventDispatcher(new KeyEventDispatcher() {
      @Override
      public boolean dispatchKeyEvent(KeyEvent e) {
        System.out.println("Key pressed: " + e.getKeyChar());
        return false;
      }
});
```

This will grab every key event and print out the printable key character. You will see that it catches the key pressed, down, and up events.

Returning false passes the event to the components as normal. If you return true then the event stops and none of the components will receive the event.

JAR Files

https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html

It is convenient to be able to group together the components of an application into a single file that can be run without have to open it as source code and compile it. A Java JAR or self-executable file is the way to do it. These files have the extension .jar and can normally be double-clicked to run the Java application.

A jar file has several benefits:

- Compression – a Jar file uses the same technology as a zip file and is compressed to reduce the size. This also puts multiple files into a single file which makes it more convenient to download.
- Security – a Jar file can be digitally signed. If the signature doesn't match then a user knows the file has been tampered with.
- Package sealing – sealing a package means that all the classes defined in that package are found in the same jar file, so all the files needed will be in one place.

Within an IDE building a Jar file is as simple as selecting "Build" from the menu. Here are the steps that happen, which we can run independently from the command line using the JDK's "jar" tool.

The format to create a jar file is:

jar cf jar-file input-file

The "cf" stands for create file. This is similar to the "tar" unix command!

For example, say we compile Test.java:

```
import javax.swing.JFrame;

public class Test extends JFrame
{
     public Test()
     {
          setSize(200,200);
          setDefaultCloseOperation(EXIT_ON_CLOSE);
          setVisible(true);
     }
    public static void main(String[] args)
    {
          Test window = new Test();
     }
}
```

We now have Test.class in the folder. Normally we would run it from the command line with "java Test"

This is not particularly convenient if you are distributing the program to others though! They don't want to go to the command line and have to run the program through Java.

To put this into a jar file we would use:

jar cf Test.jar Test.class OtherClass.class …

Note that each file is separated by a space. In our class we have only one class.

This creates a file named Test.jar. Normally it will be smaller than the source code (in our simple case it may not since there is so little code to run and not much to compress).

To run the jar file we can use:

java –jar jarfile

so:

java –jar Test.jar

If we try this we will get an error that we didn't include a manifest file. The manifest contains a signal line that points out the main class. It should look like this:

```
Main-Class:<space>Mainclass
<blank line here>
```

It is important to add the blank line or the manifest will not work! The manifest has the file suffix of .mf so normally you would create a file named manifest.mf. In our case it is as simple as:

```
Main-Class: Test
<blank line>
```

You can do other things in the manifest like seal the packages, add permissions and other security attributes, etc. You can see these in the Oracle documentation.

To build the jar file with the manifest use:

```
jar cvmf manifest.mf Test.jar Test.class
```

The “v” flag is for verbose to print diagnostic info and “m” is to include a manifest. Be careful about the order because manifest.mf must come first followed by the jar file (if we used cvfm then the jar file comes before the manifest).

The end result is a jar file that we can double-click on and it should run the application.