SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2022

Topic Notes: Keyboard Events

This brief document describes an example of using a `KeyListener` to receive and process keyboard events in a Java Swing program. You are very familiar with keyboard input to regular Java applications, using, for example, a `Scanner`. We also looked briefly at using popup windows as provided by `JOptionPane` in an earlier lab.

Here, we look at how we can handle the events generated when someone presses, releases, or types a key when our Java Swing window has *keyboard focus*. In the graphical user interfaces of most modern operating systems, the events related to keyboard activity are "delivered" to whichever program's window was most recently selected with the mouse or other pointing device. When that window is something like a Terminal, Git Bash window, your word processor, etc., that program is responsible for handling those keyboard events. When windows like the ones we've been using (without giving it much thought) process keyboard events, they respond with the familiar functionality we are used to. For example, terminals such as Git Bash or BlueJ's terminal window display what we type and package it up so our Java programs can receive it as input.

This example

`https://github.com/SienaCSISAdvancedProgramming/ArrowBall`

shows how we can set up a Java Swing window to request and process keyboard events. The process will look very familiar after all the work we've done using mouse event handlers.

The window that wishes to receive keyboard events needs to pass a reference to a class that implements the `KeyListener` interface to its `addKeyListener` method. Note that we add this to the `JFrame` rather than to the `JPanel` as we did with mouse events. As was the case with most of our mouse event handling examples, we use the same class that creates the window as our `KeyListener`, so `this` is passed as the parameter.

The `KeyListener` interface requires three methods: `keyPressed`, which is triggered each time a key is pressed or held down, `keyReleased`, which is triggered when the key is released, and `keyTyped`, which is triggered when a press-release of the same key had been completed.

In this example, we only care about `keyPressed`, so that one has a meaningful implementation. The other two are required to satisfy the interface. There is also a `KeyAdapter` class analogous to `MouseAdapter` if we only wish to override the methods we care about.

The only key presses we care about are the arrow keys. The `KeyEvent` object passed to our keyboard event handlers includes information about which key was pressed. The `KeyEvent` has a `getKeyCode` method that returns a *virtual key code*, a list of which you can find on the `KeyEvent` page in the Java documentation. We have a case in our conditional for each of the 4 arrow keys. As you can see, when we press an arrow key, the `Point` defining the location of a ball drawn in our graphics window is translated by a small amount depending on the key that was pressed. At the end, we trigger a paint event so we can see the ball in its new position.

One further note - if you are going to have multiple items in your `JFrame` that accept keyboard input (perhaps a graphics window with a `KeyListener` and a `JComboBox` or a `JTextField`, and would like to be able to use the tab key to move keyboard focus among them, you can add a call such as

```
frame.setFocusable(true);
```

to the `run` method that is responsible for creating the `JFrame` and all of the components within it.