6.005 Software Construction
Fall 2011
Prof. Rob Miller

L14: Graphical User Interfaces

Today

- View tree
- Listener pattern
- Model-view-controller pattern
- Background threads

Required reading (from the Java Tutorial)

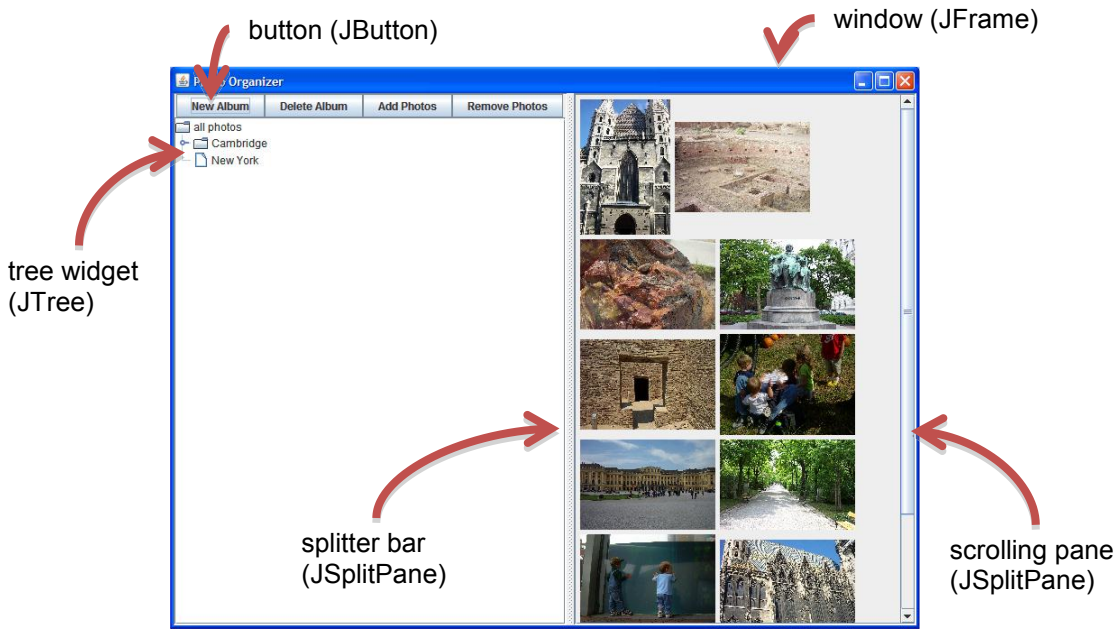
- A Visual Guide to Swing Components
- Using Swing Components, specifically:
  - Using Top-Level Containers
  - Using Text Components
  - How to Use Buttons
  - How to Use Lists
  - How to Make Frames
- Laying Out Components Within a Container, specifically:
  - How to Use GroupLayout
- Writing Event Listeners, specifically:
  - Introduction to Event Listeners
- Concurrency in Swing (up to The Event Dispatch Thread)

Today we'll take a high-level look at the software architecture of GUI software, focusing on the **design patterns** that have proven most useful. Three of the most important patterns are:

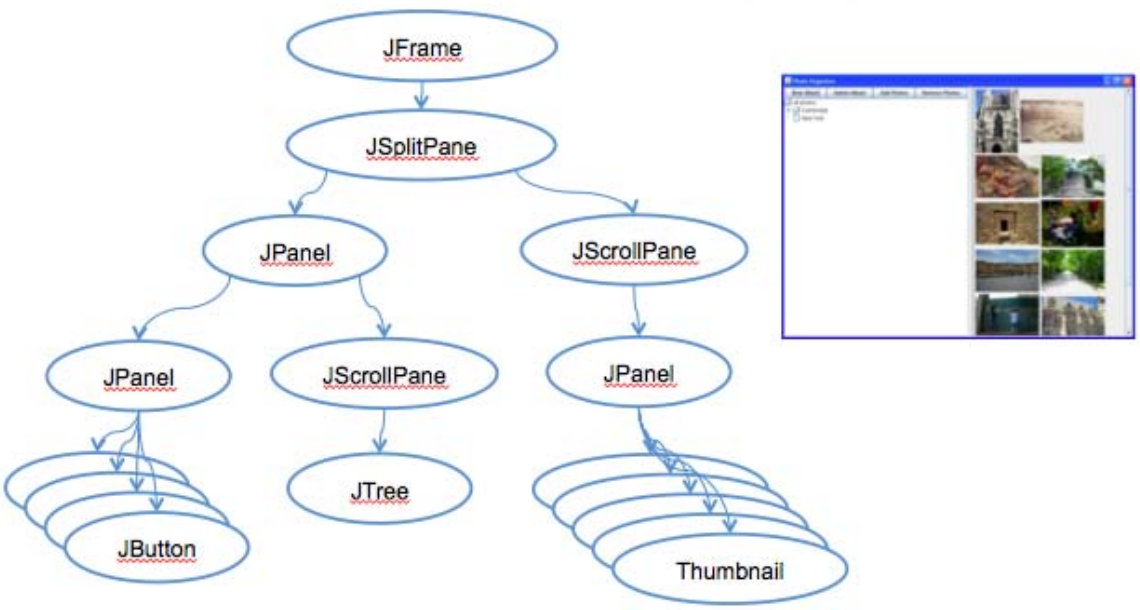
- the **model-view-controller** pattern, which has evolved somewhat since its original formulation in the early 80's;
- the **view tree**, which is a central feature in the architecture of every important GUI toolkit;
- the **listener** pattern, which is essential to decoupling the model from the view and controller.

View Tree

Graphical user interfaces are composed of **view** objects, each of which occupies a certain portion of the screen, generally a rectangular area called its *bounding box*. The view concept goes by a variety of names in various UI toolkits. In Java Swing, they're JComponents; in HTML, they're elements or nodes; in other toolkits, they may be called widgets, controls, or interactors.



This leads to the first important pattern we'll talk about today: the **view tree.** Views are arranged into a hierarchy of containment, in which some views contain other views. Typical containers are windows, panels, and toolbars. The view tree is not just an arbitrary hierarchy, but is in fact a spatial one: child views are nested inside their parent's bounding box.



View hierarchy is an example of the Composite pattern

- **Primitive views** don't contain other views
  - button, tree widget, textbox, thumbnail, etc.
- **Composite views** are used for grouping or modifying other views
  - JSplitPane displays two views side-by-side with an adjustable splitter
  - JScrollPane displays only part of a view, with adjustable scrollbars

Key idea

- primitives and composites implement a common interface (JComponent)
- containers can hold any JComponent: both primitives and other containers
- Composite pattern gives rise to a tree, with primitive views at the leaves and containers at the internal nodes

How the View Tree is Used

Virtually every GUI system has some kind of view tree. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

Output. Views are responsible for displaying themselves, and the view hierarchy directs the display process. GUIs change their output by mutating the view tree. For example, e.g., to show a new set of photos, the current Thumbnails are removed from the tree and a new set of Thumbnails is added in their place. A redraw algorithm built into the GUI toolkit automatically redraws the affected parts of the subtree. In Java Swing, this is done with the Interpreter pattern: every view in the tree has a paint() method that knows how to draw itself on the screen. The repaint process is driven by calling paint() on the root of the tree.

Input. Views can have input handlers, and the view tree controls how mouse and keyboard input is processed. More on this in a moment.

Layout. The view tree controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned. An automatic layout algorithm automatically calculates positions and sizes of views. Specialized composites (like JSplitPane, JScrollPane) do layout themselves. More generic composites (JPanel, JFrame) delegate layout decisions to a **layout manager** (e.g. GroupLayout, BorderLayout, BoxLayout, ...)

Input Handling

Input is handled somewhat differently in GUIs than we've been handling it in parsers and servers. In those systems, we've seen a single parser that peels apart the input and decides how to direct it to different modules of the program:

```
while (true) {
   read mouse click
   if (clicked on New Album) doNewAlbum();
   else if (clicked on Delete Album) doDeleteAlbum();
   else if (clicked on Add Photos) doAddPhotos();
   ...
   else if (clicked on an album in the tree) doSelectAlbum();
   else if (clicked on +/- button in the tree) doToggleTreeExpansion();
   ....
   else if (clicked on a thumbnail) doToggleThumbnailSelection();
   ...
```

}

In a GUI, we don't directly write this kind of method, because it's not modular – it mixes up responsibilities for button panel, album tree, and thumbnails all in one place. Instead, GUIs exploit the spatial separation provided by the view tree to provide functional separation as well. Mouse clicks and keyboard events are distributed around the view tree, depending on *where* they occur.

GUI input event handling is an instance of the Listener pattern (also known as Publish-Subscribe). In the Listener pattern:

- an event source generates a stream of discrete events, which correspond to state transitions in the source.
- one or more listeners register interest (subscribe) to the stream of events, providing a function to be called when a new event occurs.

In this case, the mouse is the event source, and the events are changes in the state of the mouse: its x,y position or the state of its buttons (whether they are pressed or released). Events often include additional information about the transition (such as the x,y position of mouse), which might be bundled into an **event object** or passed as parameters.

When an event occurs, the event source distributes it to all subscribed listeners, by calling their callback methods.

Control flow through a graphical user interface

- A top-level **event loop** reads input from mouse and keyboard
- For each input event, it finds the right view in the hierarchy (by looking at the x,y position of the mouse) and sends the event to that view's listeners
- Listener does its thing (e.g. modifying the view hierarchy) and returns immediately to the event loop

Higher-level GUI input events

- JButton sends an action event when it is pressed (whether by the mouse or by the keyboard)
- JTree sends a selection event when the selected element changes (whether by mouse or by keyboard)
- JTextbox sends change events when the text inside it changes for any reason

Separating Frontend from Backend

We've seen how GUI programs are structured around a view tree, and how input events are handled by attaching listeners to views. This is the start of a *separation of concerns* – output handled by views, and input handled by listeners.

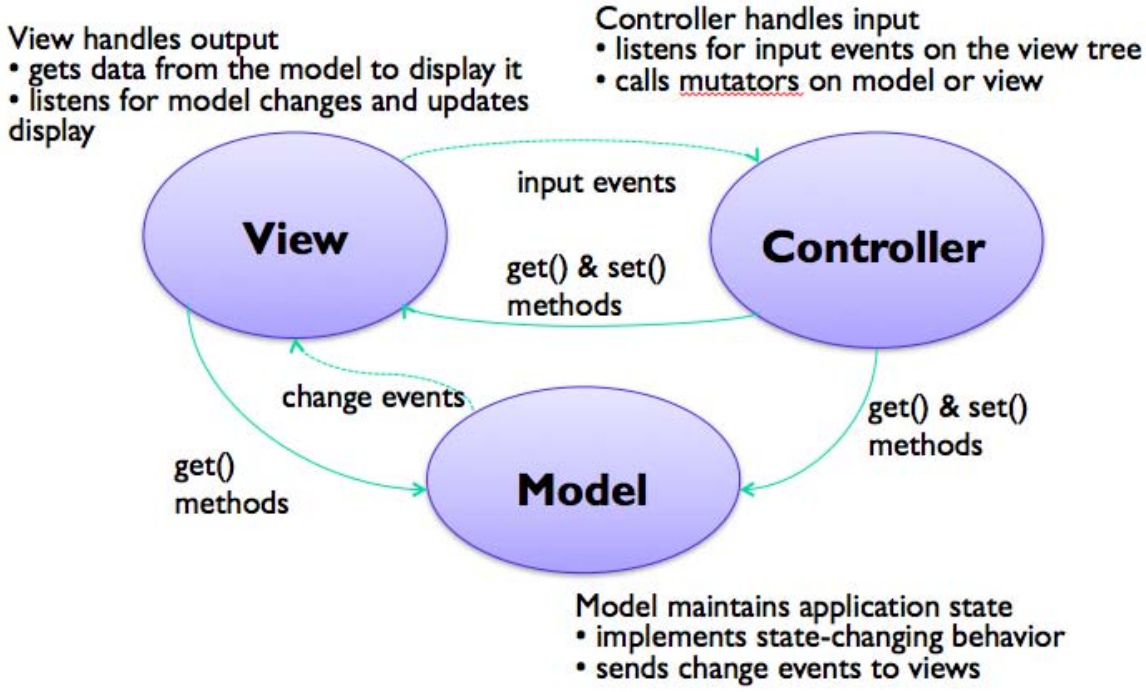
But we're still missing the application itself – the backend that represents the data and logic that the user interface is showing and editing. (Why do we want to separate this from the user interface?)

The **model-view-controller** pattern has this separation of concerns as its primary goal. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.

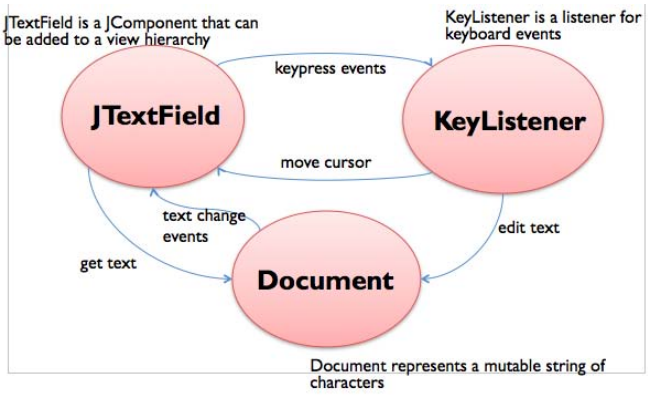
The model is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately. Models do this notification using the **listener pattern**, in which interested views and controllers register themselves as listeners for change events generated by the model.

View objects are responsible for output. A view usually occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

Finally, the controller handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

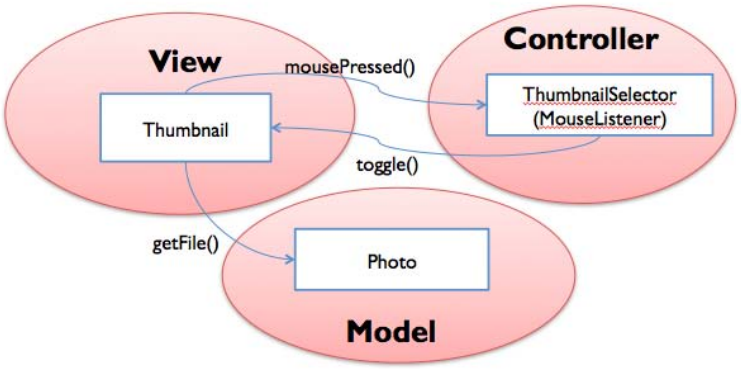


A simple example of the MVC pattern is a text field widget (this is Java Swing's text widget). Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them in the string.

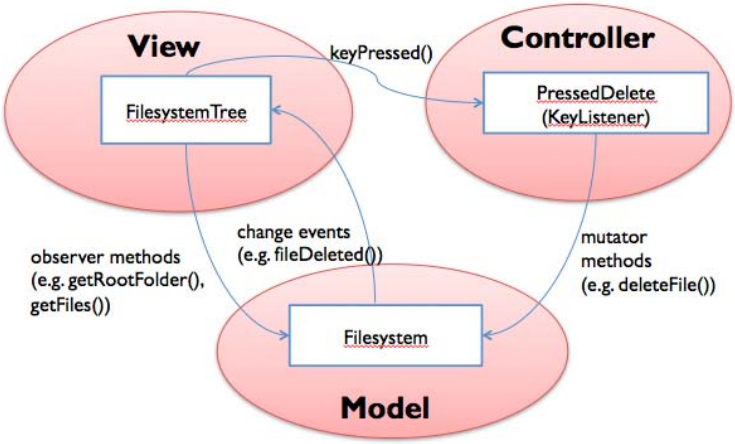


Instances of the MVC pattern appear at many scales in GUI software. At a higher level, this text field might be part of a view (like the address book editor), with a different controller listening to it (for text-changed events), for a different model (like the address book). But when you drill down to a lower level, the text field itself is an instance of MVC.

Here's an example from a photo browsing application:



And here's a larger example, in which the view is a filesystem browser (like the Mac Finder or Windows Explorer), the model is the disk filesystem, and the controller is an input handler that translates the user's keystrokes and mouse clicks into operations on the model and view.



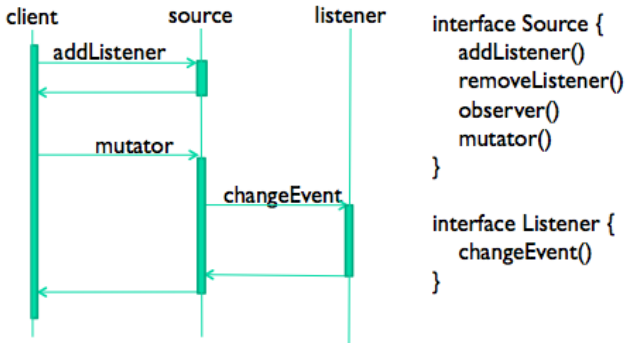
The separation of model and view has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications. The MVC pattern enables the creation of user interface **toolkits**, which are libraries of reusable views. Java Swing is such a toolkit. You can easily reuse view classes from this library (like JButton and JTree) while plugging your own models into them.

Risks of Event-Based Programming

Control flow through an event-based program is not simple. You can't follow the control just by studying the source code**,** because control flow depends on listener relationships established at runtime, and input events happening nondeterministically. Careful discipline about who listens to what (like the model-view-controller pattern) is essential for limiting the complexity of control flow and understanding how to debug your program.

The hidden control flow leads to some unexpected pitfalls, which is the next thing we'll look at in this lecture.

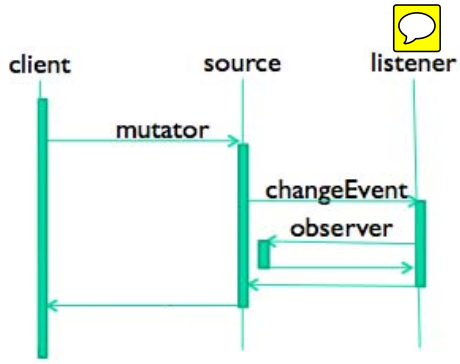
First, a bit of notation. The diagram below is a **sequence diagram**, which is useful for depicting control flow. Time flows downward. Vertical time lines represent objects, such as an event source or a listener. Horizontal arrows show method calls and returns passing control between objects. Finally, dark rectangles show when a method is active (i.e., on the call stack).



Here's the conventional interaction that occurs in the listener pattern. A client uses **addListener** (or a similar method) registers a listener to receive notifications from the event source. Then, when the source changes state (usually due to some other object calling a **mutator** method), it fires an event to all its registered listeners by calling **changeEvent** on them.

Pitfall #1: Listener Calls Observer Methods

This leads to the first pitfall. The listener often reacts to the change in the model by pulling more data from the source using observer method calls. For example, when a textbox gets a change event from its model, it needs to call getText() to get the new text and display it. So calls to observer() may occur while mutator() is still in progress.



Why is this a potential problem? Because the mutator() method hasn't returned yet, it's possible that the source data structure is not yet in a consistent state (i.e. not yet satisfying its rep invariant), which might cause the observer() method to return garbage (or worse, throw an exception).

When the source calls changeEvent() on its listeners, it is giving up control – in much the same way that a method gives up control when it returns to its caller, or that a thread gives up control when its timeslice is over and another thread takes control of the processor. In fact, what we're seeing here is a concurrency problem!

Here's some pseudocode that demonstrates this pitfall:

```
class Filesystem {
       private Map<File, List<File>> cache;
       public List<File> getContents(File folder) {
            ... check for folder in cache, else read it from disk and update
cache
    }
    public void deleteContents(File folder) {
           for (File f: getContents(folder)) {
               f.delete();
                   fireChangeEvent(f, REMOVED);  // notify listeners that f was
deleted
        }
           cache.remove(folder);  // cache is no longer valid for this folder
     }
}
```

Filesystem is a model class that represents a disk filesystem. It has an observer method getContents(), which returns the files in a folder, and a mutator method deleteContents(), which deletes all the files in a folder. To minimize disk traffic, it also maintains a cache mapping folder names to their files, so getContents() doesn't always have to hit the disk.

Now suppose a filesystem browser view is showing the files from a folder (which it obtained with getContents()) and then deleteContents() is invoked. One by one the files are removed from the folder on disk, and change events are sent back to the view – but if the view is calling getContents() to update itself, **it will always see the stale copy in the cache**. The cache isn't invalidated until the end of deleteContents(), which is too late for the view.

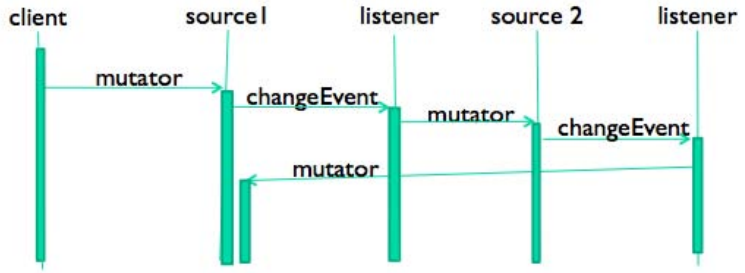
The essential problem is that this class has a rep invariant (that the cache correctly reflects the state of the filesystem) that doesn't hold when the view calls getContents(). It's quite normal, even inevitable, for invariants to be *temporarily* unsatisfied while a mutator method is executing. In this case, event-passing has created an opportunity for a client to get control during that period of inconsistency, and see buggy results.

So an event source has to make sure that it's consistent --- i.e., that it has established all of its internal invariants – before it starts issuing notifications to listeners. It's often best to delay firing off events until the end of the method that caused the modification. Don't fire events while you're in the midst of making changes to the model's data structure. This example could be fixed either by

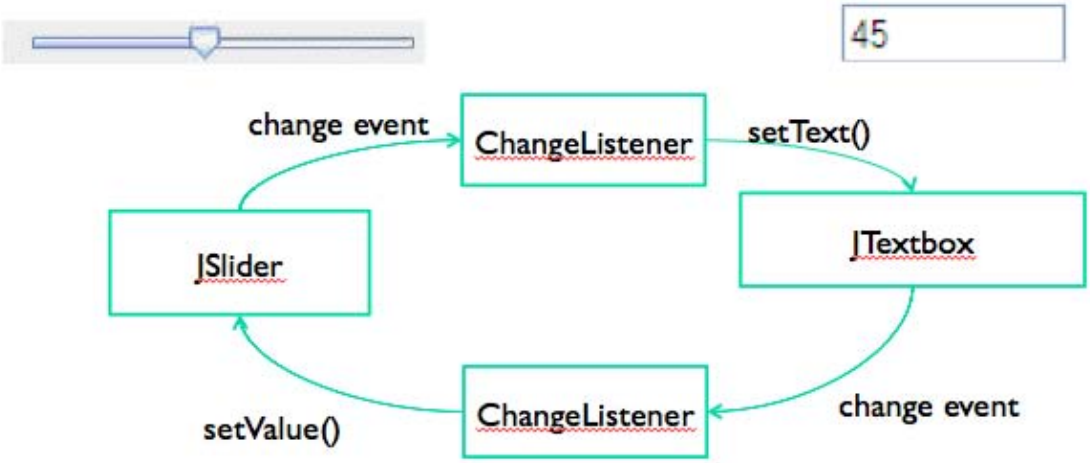
batching up events until the end of deleteContents(), or by invalidating the cache *before* starting to remove files, so that the invariant is never unsatisfied.

Pitfall #2: Listener Calls Mutator Methods

Another pitfall occurs when a listener responds to an update message by calling the mutator on the model. Why would it do that? It might, for instance, be trying to keep the model within some legal range. Or two models could be listening to each other in order to keep their state synchronized. So recursive calls to mutator() may occur while mutator() is still in progress. Obviously, this could lead to infinite regress if you're not careful.



Here's a concrete example of this pitfall: a numeric slider and a textbox, which you want to synchronize with each other so that the user can change the value using either widget:

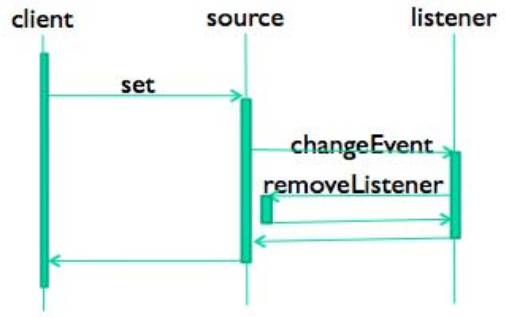


So both widgets end up listening to each other's changes, and an infinite regress might result.

A good practice for models to protect themselves against this regress is to only send update events if a change actually occurs; if a client calls mutator() but it has no actual effect on the model, then no events should be sent.

Pitfall #3: Listener Removes Itself

Another potential pitfall is a listener that unregisters itself with **removeListener**. For example, suppose we have a model of stock market data, and a listener that's watching for a certain stock to reach a certain price. Once the stock hits the target price, the listener does its thing (e.g., popping up a window to notify the user, or executing a trade); but then it's no longer needed, so it unregisters itself from the model.



This is a problem if the model is iterating naively over its collection of listeners, and the collection is allowed to change in the midst of the iteration. Here's some concrete code showing the problem – the listeners are stored as a simple fixed-size array. Trace through and see what happens if the ith listener calls removeListener() on itself from its changeEvent() method:

```
class Source {
   private Listener[] listeners;
   private int size;
   public void removeListener(Listener l) {
       for (int i = 0; i < size; ++i) {
           if (listeners[i] == l) {
               listeners[i] = listeners[size-1]; --size;
           }
       }
   }
   private void fireChangeEvent(...) {
      for (int i = 0; i < size; ++i) {
          listeners[i].changeEvent(...);
      }
   }
}
```

Most Java collections (Lists, Sets) have the same problem. If you're lucky, Java will throw a ConcurrentModificationException when you mutate a collection that you're currently iterating. If you're not, your list will simply be quietly corrupted.

It's safer to iterate over a *copy* of the listener list. Since one-shot listeners are not particularly common, however, this imposes an extra cost on every event broadcast. So the ideal solution is to copy the listener list only when necessary – i.e., when a register or unregister occurs in the midst of event dispatch. javax.swing.EventListenerList works this way.

Background Processing in Graphical User Interfaces

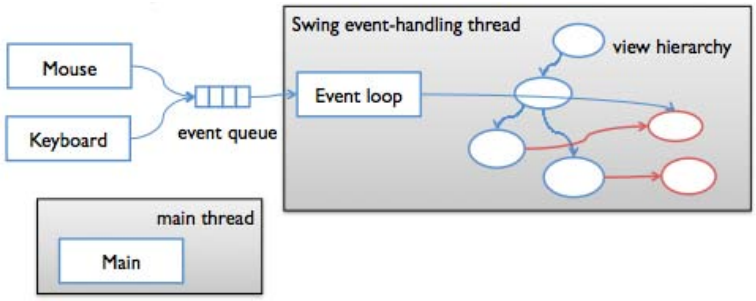
The last major topic for today connects back to concurrency.

First, some motivation. Why do we need to do background processing in graphical user interfaces? Even though computer systems are steadily getting faster, we're also asking them to do more. Many programs need to do operations that may take some time: retrieving URLs over the network, running database queries, scanning a filesystem, doing complex calculations, etc.

But graphical user interfaces are event-driven programs, which means (generally speaking) **everything is triggered by an input event handler**. For example, in a web browser, clicking a hyperlink starts loading a new web page. But if the click handler is written so that it actually retrieves the web page itself, then the web browser will be very painful to use. Why? Because its interface will appear to **freeze up** until the click handler finishes retrieving the web page and returns to the event loop. Here's why.

This happens because input handling and screen repainting is all handled from a single thread. That thread (called the **event handling thread**) has a loop that reads an input event from the queue and dispatches it to listeners on the view hierarchy. When there are no input events left to process, it repaints the screen. But if an input handler you've written delays returning to this loop – because it's blocking on a network read, or because it's searching for the solution to a big Sudoku puzzle – then input events stop being handled, and the screen stops updating. So long tasks need to run in the background.

In Java, the event-handling thread is distinct from the main thread of the program (see below). It is started automatically when a user interface object is created. As a result, every Java GUI program is **automatically multithreaded**. Many programmers don't notice, because the main thread typically doesn't do much in a GUI program – it starts creation of the view, and then the main thread just exits, leaving only the event-handling thread to do the main work of the program.



The fact that Swing programs are multithreaded by default creates risks. There's very often a shared mutable datatype in your GUI: the **model**. If you use background threads to modify the model without blocking the event-handling thread, then you have to make sure your data structure is threadsafe.

But another important shared mutable datatype in your GUI is the view tree. **Java Swing's view tree is not threadsafe**. In general, you cannot safely call methods on a Swing object from anywhere but the event-handling thread.

The view tree is a big meatball of shared state

- And there's no lock protecting it at all
- It's confined (by specification) to the event-handling thread, so it's ok to access view objects from the event-handling thread (i.e., in response to input events)
- But the Swing specification forbids touching – reading *or* writing – any JComponent objects from a different thread
  - See "Threads and Swing", http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html
  - The truth is that Swing's implementation does have *one big lock* (Component.getTreeLock()) but only some Swing methods use it (e.g. layout)

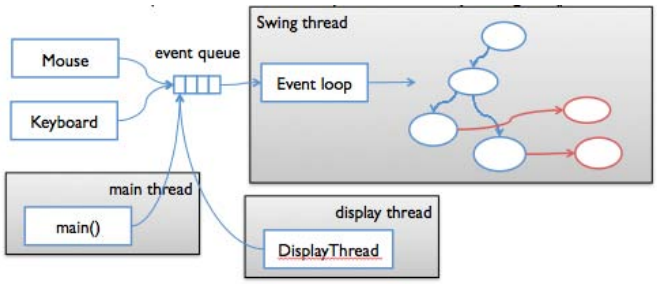
Solution: the event queue is also a message-passing queue

- To access or update Swing objects from a different thread, you can put a message (represented as a Runnable object) on the event queue

```
SwingUtilities.invokeLater(new Runnable() {
       public void run() {
           content.add(thumbnail);
           ...
```

```
        }
});
```

- The event loop handles one of these pseudo-events by calling run()



Summary

View hierarchy

- Organizes the screen into a tree of nested rectangles
- Used for dispatching input as well as displaying output
- Uses the Composite pattern: compound views (windows, panels) can be treated just like primitive views (buttons, labels)

Publish-subscribe pattern

- An event source sends a stream of events to registered listeners
- Decouples the source from the identity of the listeners
- Beware of pitfalls

MVC pattern

- Separation of responsibilities: model=data, view=output, controller=input
- Decouples view from model

Background threads

- Swing view tree is confined to the event-handling thread
- To read and write the tree from another thread, use the event loop as a message-passing queue

6.005 Elements of Software Construction

Fall 2011