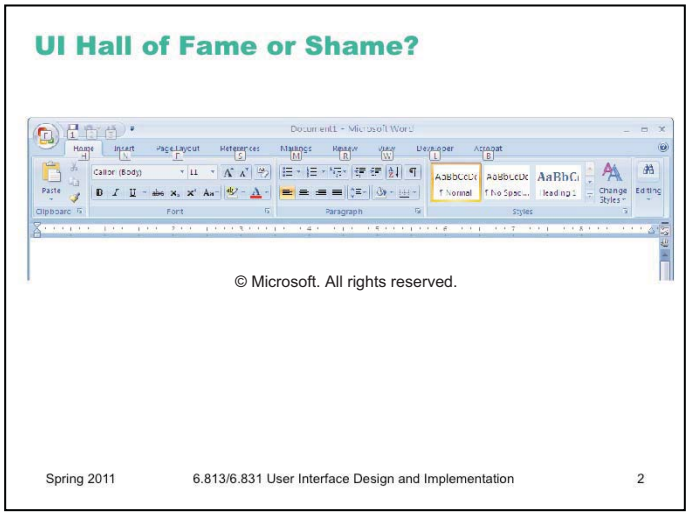
Lecture 10: Layout



Our Hall of Fame or Shame candidate for today is the command ribbon, which was introduced in Microsoft Office 2007. The ribbon is a radically different user interface for Office, merging the menubar and toolbars together into a single common widget. Clicking on one of the tabs ("Home", "Insert", "Page Layout", etc) switches to a different ribbon of widgets underneath. The metaphor is a mix of menubar, toolbar, and tabbed pane. Notice how UIs have evolved to the point where new metaphorical designs are riffing on existing *GUI* objects, rather than *physical* objects. Expect to see more of that in the future.

Needless to say, strict **external consistency** has been thrown out the window – Office no longer has a menubar or toolbar. But if we were slavishly consistent, we'd never make any progress in user interface design. Despite the radical change, the ribbon *is* still externally consistent in some interesting ways, with other Windows programs and with previous versions of Office. If you look carefully at the interface, they *are* consistent in some important ways: (1) critical toolbar buttons still look the same, like Save, Cut, Copy, and Paste; (2) the command tab buttons resemble menubar menus, in both location and naming; (3) the ribbons look and act like rich toolbars, with familiar widgets and familiar affordances. So even though some new learning has to happen for Office 12 users, the knowledge transfer from other apps or previous Office is likely to be substantial.

One thing Office 12's developers did very effectively is **task analysis**. In fact, they signed up thousands of Office users to a special program that collected statistics on how frequently they used Office commands and in which order – huge amounts of data that directly drove how commands were grouped into the command tabs, and which commands appear on command tabs as opposed to being buried in deeper dialogs. When a user interface designer can get this kind of data, you can do a lot to improve the usability for an average user. Web site designers are lucky, in this sense, because server logs give it to them for free! Microsoft had to do a lot more work to get it.

Office 2007 also provides more **feedback** about what a command will do, by showing a preview of its effect right in the document while you're mousing over the command. So if you hover over the Heading 2 option, your document will reformat to show you what the selection would look like with that new style. As long as your computer is fast enough to do it within 100ms, this would be a tremendous improvement to the visibility and feedback of the interface.

Today's Topics

- CSS
- Automatic layout
- Constraints

Today's lecture is about **automatic layout** – determining the positions and sizes of UI components. Automatic layout is an good example of declarative user interface specification. The programmer specifies what kind of layout is desired by attaching properties or layout managers to the view hierarchy, and then an automatic algorithm (layout propagation) actually computes the layout.

We'll also talk about **constraints**, which is a rather low-level, but also declarative, technique for specifying layout. Constraints are useful for more than just layout; unfortunately most GUI toolkits don't have a general-purpose constraint solver built in. But constraints are nevertheless a useful way to think about relationships in a user interface declaratively, even if you have to translate them to procedural code yourself.

Cascading Style Sheets (CSS)

- Key idea: separate the **structure** of the UI (view tree) from details of **presentation**
  - HTML is structure, CSS is presentation
- Two ways to use CSS
  - As an attribute of a particular HTML element

    <button **style="font-weight:bold;"**> Cut </button>
  - As a style sheet defining style rules for many HTML elements at once

    <style>

    button { font-weight:bold; }

    </style>

Our second example of declarative specification is Cascading Style Sheets, or CSS. Where HTML creates a view hierarchy, CSS adds style information to the hierarchy – fonts, colors, spacing, and layout.

There are two ways to use CSS. The first way is by setting styles directly on individual objects. The style attribute of any HTML element can contain a set of CSS settings (which are simply **name:value** pairs separated by semicolons).

The second way is more interesting, because it's more declarative. Rather than finding each individual component and directly setting its style attribute, you specify a **style sheet** that defines rules for assigning styles to elements. Each rule consists of a pattern that matches a set of HTML elements, and a set of CSS definitions that specify the style for those elements. In this simple example, **button** matches all the button elements, and the body of the rule sets them to boldface font.

The style sheet is included in the HTML by a <style> element, which either embeds the style sheet as text between <style> and </style>, or refers to a URL that contains the actual style sheet.

CSS Selectors

- Each rule in a style sheet has a **selector** pattern that matches a set of HTML elements

Tag name

```
button { font-weight:bold; }
```

ID

```
#main { background-color:
        rgb(100%,100%,100%); }
```

Class attribute

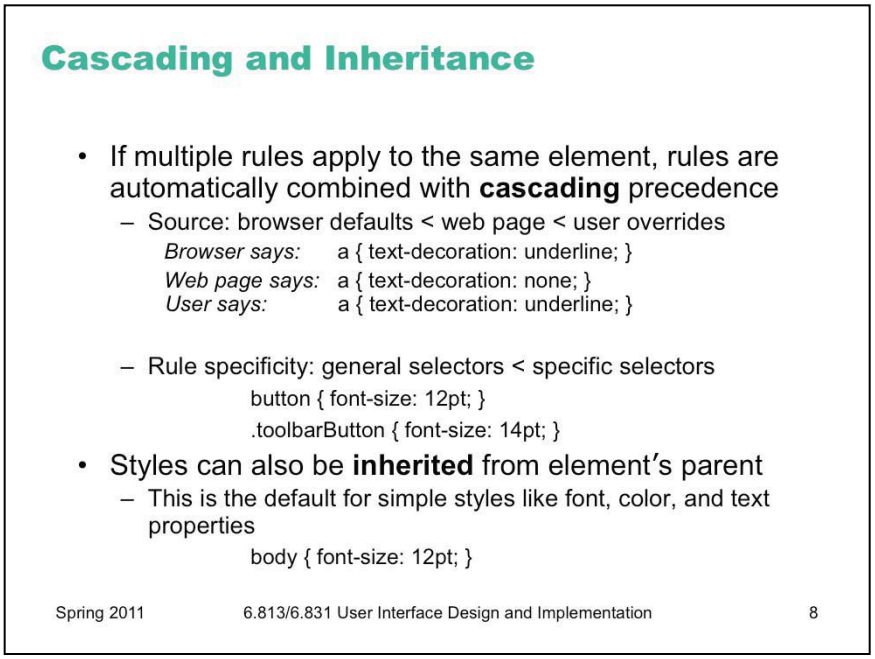
```
.toolbarButton { font-size: 12pt; }
```

Element paths

```
#toolbar button { display: hidden; }
```

```
<div id="main">
  <div id="toolbar">
    <button class="toolbarButton">
      <img src="cut.png"></img>
    </button>
  </div>
  <textarea id="editor"></textarea>
</div>
```


The pattern in a CSS rule is called a **selector**. The language of selectors is simple but powerful. Here are a couple of the more common selectors. Selectors are also used by jQuery to select and operate on nodes in the DOM tree, so it's worth becoming familiar with this pattern language.



There can be multiple style sheets affecting an HTML page, and multiple rules within a style sheet. Each rule affects a set of HTML elements, so what happens when an element is affected by more than one rule? If the rules specify independent style properties (e.g., one rule specifies font size, and another specifies color), then the answer is simple: both rules apply. But what if the rules *conflict* with each other – e.g., one says the element should be bold, and another says it shouldn't?

To handle these cases, declarative rule-based systems need a conflict resolution mechanism, and CSS is no different. CSS's resolution mechanism is called **cascading** (hence the name, Cascading Style Sheets). It has two main resolution strategies. The overall idea is that more specific rules should take precedence over more general rules. This is reflected first in where the style sheet rule came from: some rules are web browser defaults, for all users and all web pages; others are defaults set by a specific user for all web pages; others are provided by a specific web page in a <style> element. In general, the web page rule wins (although the user can override this by setting the priority of their own CSS rules to *important*). Second, rules with more specific selectors (like specific element IDs or class names) take precedence over rules with more general selectors (like element names).

This is an example of why declarative specification is powerful. A single rule – like a user override – can affect a large swath of the behavior of the system, without having to write a lot of procedural code, and without having to make sure that procedural code runs at just the right time.

But it also illustrates the difficulties of debugging declarative specifications. You may add a rule to the style sheet, maybe trying to change a button's font size, only to see *no change* in the result – because some other rule that you aren't aware of is taking precedence. CSS conflict resolution is a complex process that may require trial-and-error to debug.

Declarative Styles vs. Procedural Styles

CSS

```
// found in a <style> element
button { font-size: 12pt; font-weight: bold; }
```

jQuery

```
// in a <script> element
$("button").css("font-size", "12pt").css("font-weight", "bold");
```


Just as with HTML, we can change CSS styles procedurally as well. jQuery offers a particularly nice way to do this, which matches very closely the parts of a CSS rule: a selector, a property name, and a value.

Automatic Layout

- **Layout** determines the sizes and positions of components on the screen
  - Also called geometry in some toolkits
- Declarative layout
  - CSS styles
- Procedural layout
  - Write Javascript code to compute positions and sizes

In HTML/CSS, automatic layout is a declarative process. First you specify the graphical objects that should appear in the window, which you do by creating instances of various objects and assembling them into a view tree. We've seen how HTML does this. Then you specify how they should be laid out by attaching styles.

You can contrast this to a procedural approach to layout, in which you write Javascript code that computes positions and sizes of objects in the view tree.

Reasons to Do Automatic Layout

- Higher level programming
  - Shorter, simpler code
- Adapts to change
  - Window size
  - Font size
  - Widget set (or theme or skin)
  - Labels (internationalization)
  - Adding or removing nodes

Here are the two key reasons why we like automatic layout – and these two reasons generalize to other forms of declarative UI as well.

First, it makes programming **easier**. The code that sets up layout managers is usually much simpler than procedural code that does the same thing.

Second, the resulting layout can **respond to change** more readily. Because it is generated automatically, it can be *regenerated* any time changes occur that might affect it. One obvious example of this kind of change is resizing the window, which increases or decreases the space available to the layout. You could handle window resizing with procedural code as well, of course, but the difficulty of writing this code means that programmers generally *don't*. (That's why many Windows dialog boxes, which are often laid out using absolute coordinates in a GUI builder, refuse to be resized! A serious restriction of user control and freedom, particularly if the dialog box contains a list or file chooser that would be easier to use if it were larger.)

Automatic layout can also automatically adapt to font size changes, different widget sets (e.g., buttons of different size, shape, or decoration), and different labels (which often occur when you translate an interface to another language, e.g. English to German). These kinds of changes tend to happen as the application is moved from one platform to another, rather than dynamically while the program is running; but it's helpful if the programmer doesn't have to worry about them.

Another dynamic change that automatic layout can deal with is the appearance or disappearance of nodes from the view tree-- if the user is allowed to add or remove buttons from a toolbar, for example, or if new textboxes can be added or removed from a search query.

Flow Layout

- Left-to-right, automatically-wrapping
- CSS calls this “inline” layout
  display: inline
- Many elements use inline layout by default

```
<button>People</button>
<button>Places</button>
<button>Things</button>
<button>New</button>
<button>Save</button>
<button>Print</button>
…
```

<button>People</button>

<button>Places</button>

<button>Things</button>

<button>New</button>

<button>Save</button>

<button>Print</button>

Author:

<input type="text" />

Comment:

<textarea></textarea>

<button>OK</button>

<button>Cancel</button>

Block Layout

- Blocks are laid out vertically
  - display: block
  - divs default to block layout
- Inline blocks are laid out in flow
  - display: inline-block

```
<div>
<button>People</button>
<button>Places</button>
<button>Things</button>
</div>
<div>
<button>New</button>
<button>Save</button>
<button>Print</button>
</div>
```

Float Layout

- Float pushes a block to left or right edge

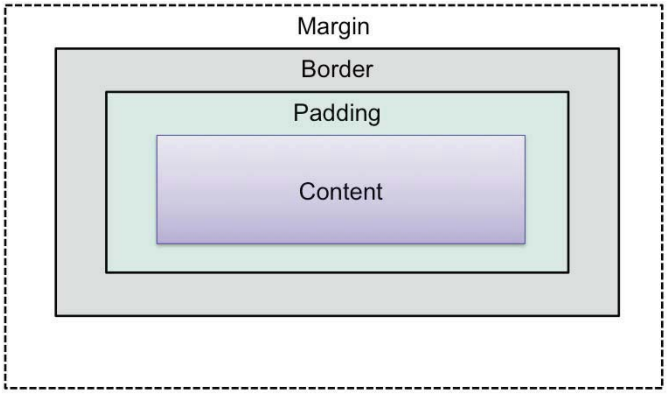
```
<style>
.navbar { float: left; }
.navbar button { display: block; }
</style>

<div class="navbar">
<button>People</button>
<button>Places</button>
<button>Things</button>
</div>

<div>
<button>New</button>
<button>Save</button>
<button>Print</button>
</div>
```

Grid Layout

- Blocks & floats are typically not enough to enforce all the alignments you want in a UI
- Use tables instead

```
<table>
<tr><td>Name:</td>
      <td><input type="text" /></td>
</tr>
<tr><td>Groups:</td>
      <td><textarea></textarea></td>
</tr>
</table>
```

Margins, Borders, & Padding



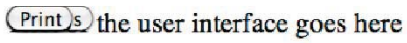
Space-Filling & Alignment

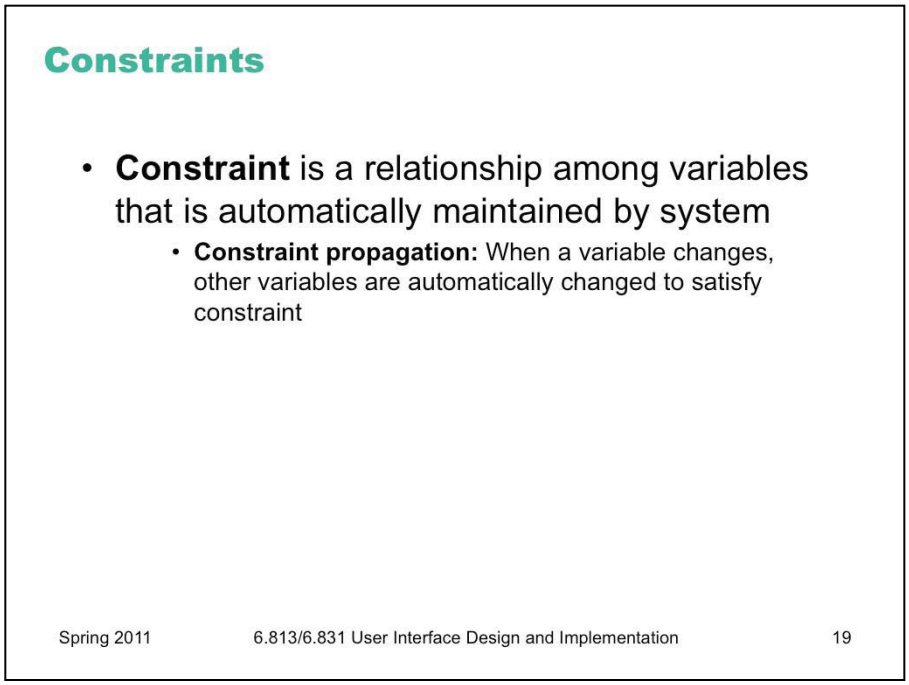
- width: 100% , height: 100% consumes all of the space available in the parent
- vertical-align moves a node up and down in its parent's box
  - baseline is good for lining up labels with textboxes
  - top and bottom are useful for other purposes
- Centering
  - margin: auto for boxes
  - text-align: center for inlines

Absolute Positioning

- Setting position & size explicitly
  - in coordinate system of entire window, or of node's parent
  - CSS has several units: px, em, ex, pt
  - mostly useful for popups

```
<style>
button { position: absolute;
         left: 5px;
         top: 5px; }
</style>
```

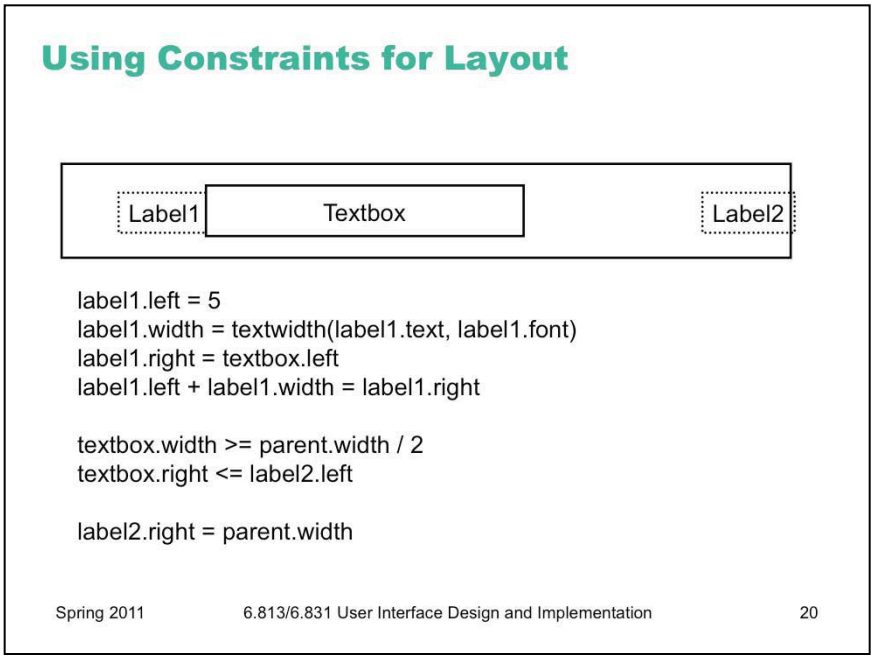




Since CSS layout has limitations, let's look at a more general form of declarative UI, that can be used not only for layout but for other purposes as well: **constraints**. HTML/Javascript doesn't support them, but other toolkits do to some extent (notably Adobe Flash).

A constraint is a relationship among variables. The programmer specifies the relationship, and then the system tries to automatically satisfy it. Whenever one variable in the constraint changes, the system tries to adjust variables so that the constraint continues to be true. Constraints are rarely used in isolation; instead, the system has a collection of constraints that it's trying to satisfy, and a **constraint propagation** algorithm satisfies the constraints when a variable changes.

In a sense, layout managers are a limited form of constraint system. Each layout manager represents a set of relationships among the positions and sizes of the children of a single container; and layout propagation finds a solution that satisfies these relationships.



Here's an example of some constraint equations for layout. This is same layout we showed a couple of slides ago, but notice that we didn't need struts or springs here; constraint equations can do the job instead.

This simple example reveals some of the important issues about constraint systems. One issue is whether the constraint system is **one-way** or **multiway**. One-way constraint systems are like spreadsheets – you can think of every variable like a spreadsheet cell with a formula in it calculating its value in terms of other variables. One-way constraints must be written in the form X=f (X1,X2,X3,…). Whenever one of the Xi's changes, the value of X is recalculated. (In practice, this is often done **lazily** – i.e., the value of X isn't recalculated until it's actually needed.)

Multiway constraints are more like systems of equations -- you could write each one as f(X1,X2,X3, …) = 0. The programmer doesn't identify one variable as the output of the constraint – instead, the system can adjust any variable (or more than one variable) in the equation to make the constraint become true. Multiway constraint systems offer more declarative power than one-way systems, but the constraint propagation algorithms are far more complex to implement.

One-way constraint systems must worry about **cycles**: if variable X is computed from variable Y, but variable Y must be computed from variable X, how do you compute it? Some systems simply disallow cycles (spreadsheets consider them errors, for example). Others break the cycle by reusing the old (or default) value for one of the variables; so you'll compute variable Y using X's old value, then compute a new value for X using Y.

Conflicting constraints are another problem – causing the constraint system to have no solution. Conflicts can be resolved by **constraint hierarchies**, in which each constraint equation belongs to a certain priority level. Constraints on higher priority levels take precedence over lower ones.

Inequalities (such as textbox.right <= label2.left) are often useful in specifying layout constraints, but require more expensive constraint satisfaction algorithms.

Using Constraints for Behavior

- Input
  - checker.(x,y) = mouse.(x,y)
    if mouse.button1 && mouse.(x,y) in checker
- Output
  - checker.dropShadow.visible = mouse.button1 && mouse.(x,y) in checker
- Interactions between components
  - deleteButton.enabled = (textbox.selection != null)
- Connecting view to model
  - checker.x = board.find(checker).column * 50

Constraints can be used for more general purposes than just layout. Here are a few.

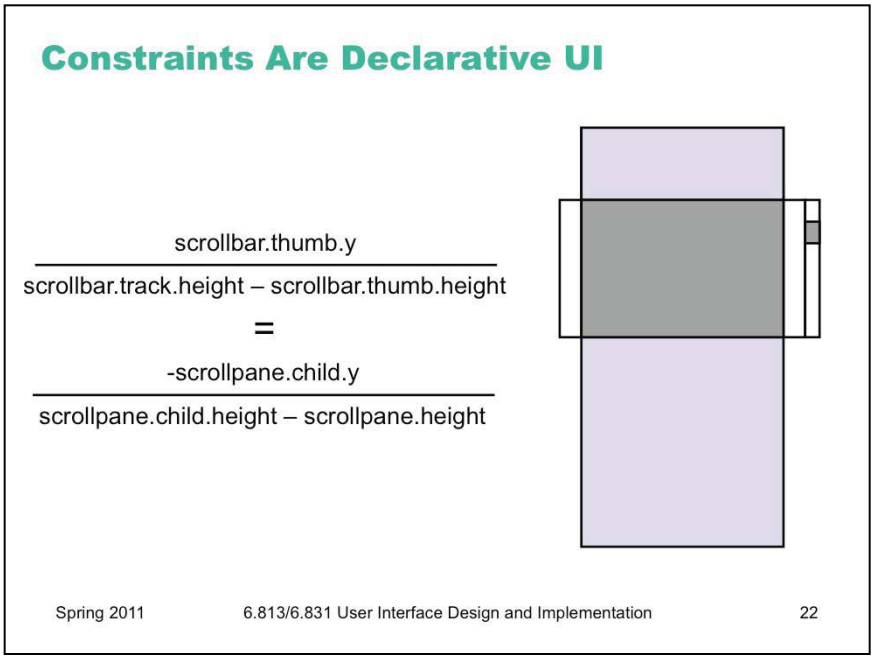
Some forms of **input** can be handled by constraints, if you represent the state of the input device as variables in constraint equations. For example, to drag a checker around on a checkerboard, you constrain its position to the position of the mouse pointer.

Constraints can be very useful for keeping user interface components consistent with each other. For example, a Delete toolbar button and a Delete command on the Edit menu should only be enabled if something is actually selected. Constraints can make this easy to state.

The connection between a view and a model is often easy to describe with constraints, too. (But notice the conflicting constraints in this example! checker.x is defined both by the dragging constraint and by the model constraint. Either you have to mix both constraints in the same expression – e.g., if dragging then use the dragging constraint, else use the model constraint – or you have to specify priorities to tell the system which constraint should win.)

The alternative to using constraints in all these cases is writing **procedural code** – typically an event handler that fires when one of the dependent variables changes (like mouseMoved for the mouse position, or selectionChanged for the textbox selection, or pieceMoved for the checker position), and then computes the output variable correctly in response. The idea of constraints is to make this code **declarative** instead, so that the system takes care of listening for changes and computing the response.

Flex's bindings can be used this way.



This example shows how powerful constraint specification can be. It shows how a scrollbar's thumb position is related to the position of the pane that it's scrolling. (The pane's position is relative to the coordinate system of the scroll window, which is why it's *negative*.) Not only is it far more compact than procedural code would be, but it's **multiway.** You can solve this equation for different variables, to compute the position of the scrollpane as a function of the thumb position (in order to respond to the user dragging the thumb), or to compute the thumb position as a function of the pane position (e.g. if the user scrolls the pane with arrow keys or jumps directly to a bookmark). So both remain consistent.

Alas, constraint-based user interfaces are still an area of research, not much practice. Some research UI toolkits have incorporated constraints (Amulet, Artkit, Subarctic, among others), and a few research constraint solvers exist that you can plug in to existing toolkits (e.g., Cassowary). But you won't find constraint systems in most commercial user interface toolkits, except in limited ways. The SpringLayout layout manager is the closest thing to a constraint system you can find in standard Java (it suffers from the limitations of all layout managers).

But you can still *think* about your user interface in terms of constraints, and *document your code* that way. You'll find it's easier to generate procedural code once you've clearly stated **what** you want (declaratively). If you state a constraint equation, then you know which events you have to listen for (any changes to the variables in your equation), and you know what those event handlers should do (solve for the other variables in the equation). Writing procedural code for the scrollpane is much easier if you've already written the constraint relationship.

Summary

- Automatic layout adapts to different platforms and UI changes
- CSS for layout
- Constraints are useful for more than just layout

6.831 / 6.813 User Interface Design and Implementation

Spring 2011