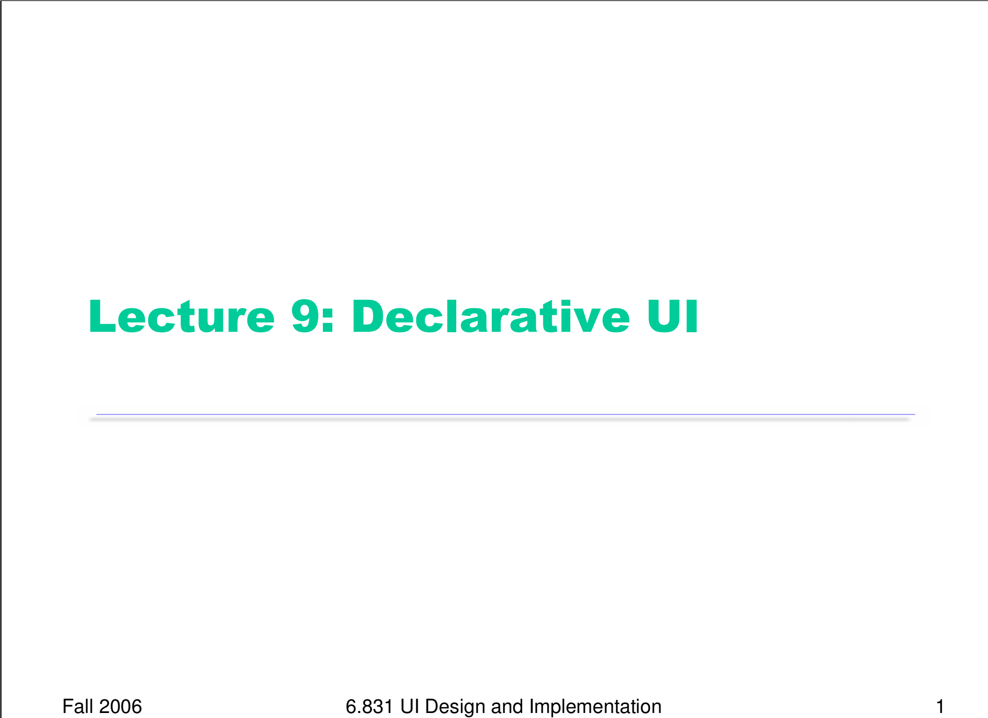
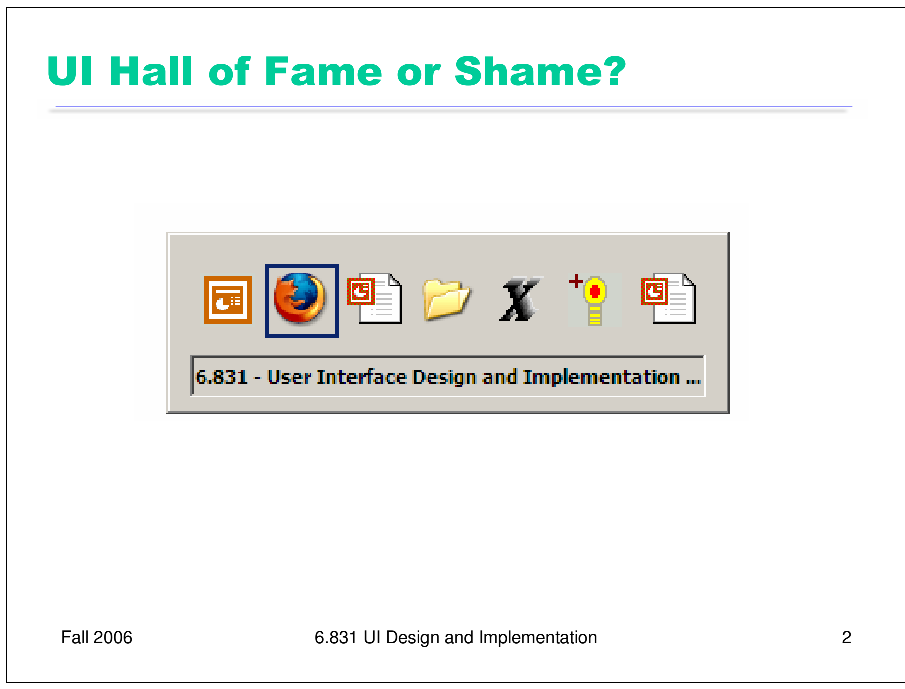

Lecture 9: Declarative UI
Fall 2006
6.831 UI Design and Implementation
1



Today's candidate for the Hall of Fame & Shame is the **Alt-Tab** window switching interface in Microsoft Windows. This interface has been copied by a number of desktop systems, including KDE, Gnome, and even Mac OS X.

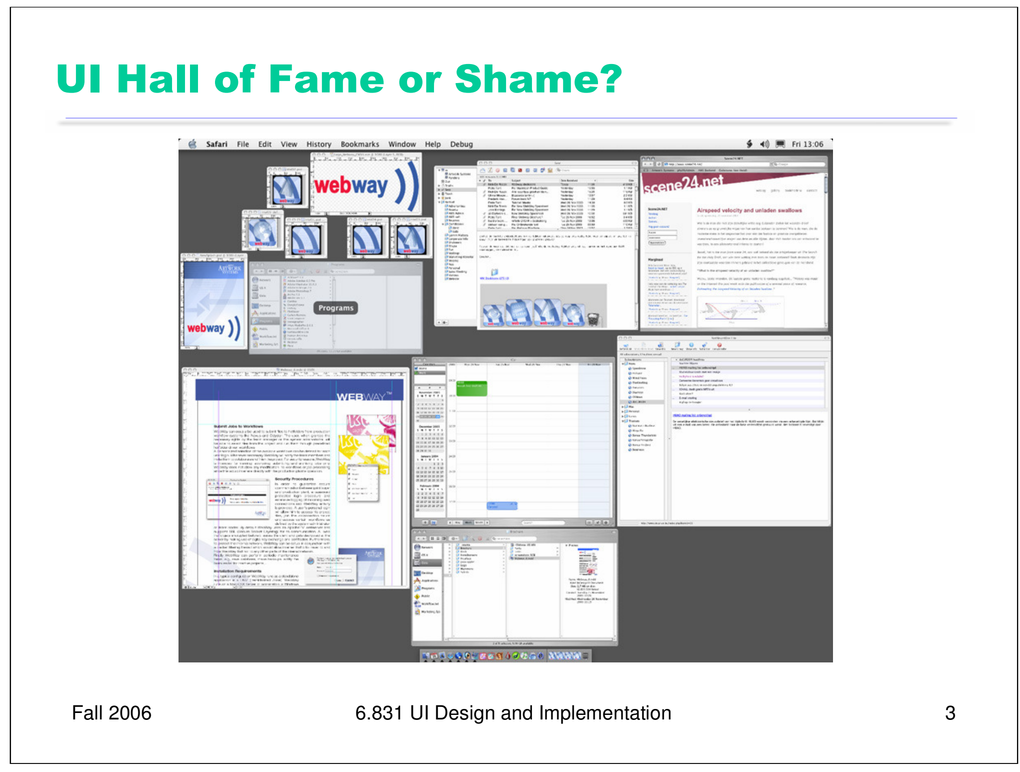
The first observation to make is that this interface is designed only for keyboard interaction. Alt-Tab is the only way to make it appear; pressing Tab (or Shift-Tab) is the only way to cycle through the choices. If you try to click on this window with the mouse, it vanishes. The interface is weak on affordances, and gives the user little help in remembering how to use it.

But that's OK, because the Windows taskbar is the primary interface for window switching, providing much better visibility and affordances. This Alt-Tab interface is designed as a **shortcut**, and we should evaluate it as such.

It's pleasantly **simple**, both in graphic design and in operation. Few graphical elements, good alignment, good balance. The 3D border around the window name could probably be omitted without any loss.

This interface is a **mode** (since pressing Tab is switching between windows rather than inserting tabs into text), but it's spring-loaded, happening only as long as the Alt button is held down. This spring-loading also provides good **dialog closure**.

Is it **efficient**? A common error, when you're tabbing quickly, is to overshoot your target window. You can fix that by cycling around again, but that's not as **reversible** as just moving backwards with a mouse. (You can also back up by holding down Shift when you press Tab, but that's not well-communicated by this interface, and it's tricky to negotiate while you're holding Alt down.)



For comparison, let's look at the Exposé feature in Mac OS X. When you push F9 on a Mac, it displays all the open windows – even hidden windows, or windows covered by other windows – shrinking them as necessary so that they don't overlap. Mousing over a window displays its title, and clicking on a window brings that window to the front and ends the Exposé mode, sending all the other windows back to their old sizes and locations.

Like Alt-Tab, Exposé is also a **mode**. Unlike Alt-Tab, however, it is not spring-loaded. It depends instead on dramatic visual differences as a mode indicator – with its shrunken, tiled windows, Exposé mode usually looks a lot different than the normal desktop.

To get out of Exposé mode *without* choosing a new window, you can press F9 again, or you can click the window you were using before. That's easier to discover and remember than Alt-Tab's mechanism – pressing Escape. When I use Alt-Tab, and then decide to abort it, I often find myself cycling through all the windows trying to find my original window again. Both interfaces support **user control and freedom**, but Exposé seems to make canceling more **efficient**.

The representation of windows is much richer in Exposé than in Alt-Tab. Rather than Alt-Tab's icons (many of which are identical, when you have several documents open in the same application), Exposé uses the **window itself** as its visual representation. That's much more in the spirit of direct manipulation. (A version of Alt-Tab included in Windows Power Toys shows images of the windows themselves – try it!)

Let's look at efficiency more deeply. Alt-Tab is a very linear interface – to pick an arbitrary window out of the n windows you have open, you have to press Tab O(n) times. Exposé, on the other hand, depends on pointing – so because of Fitts's Law, the cost is more like O(log n). (Of course, this analysis only considers motor movement, not visual search time; it assumes you already know where the window you want is in each interface. But Exposé probably wins on visual search, too, since the visual representation shows the window itself, rather than a frequently-ambiguous icon.)

But Alt-Tab is designed to take advantage of **temporal locality;** the windows you visited recently are at the start of the list. So even if Exposé is faster at getting to an arbitrary window, Alt-Tab really wins on one very common operation: toggling back and forth between two windows.

Today's Topics

- Declarative user interface
- Automatic layout
- Constraints

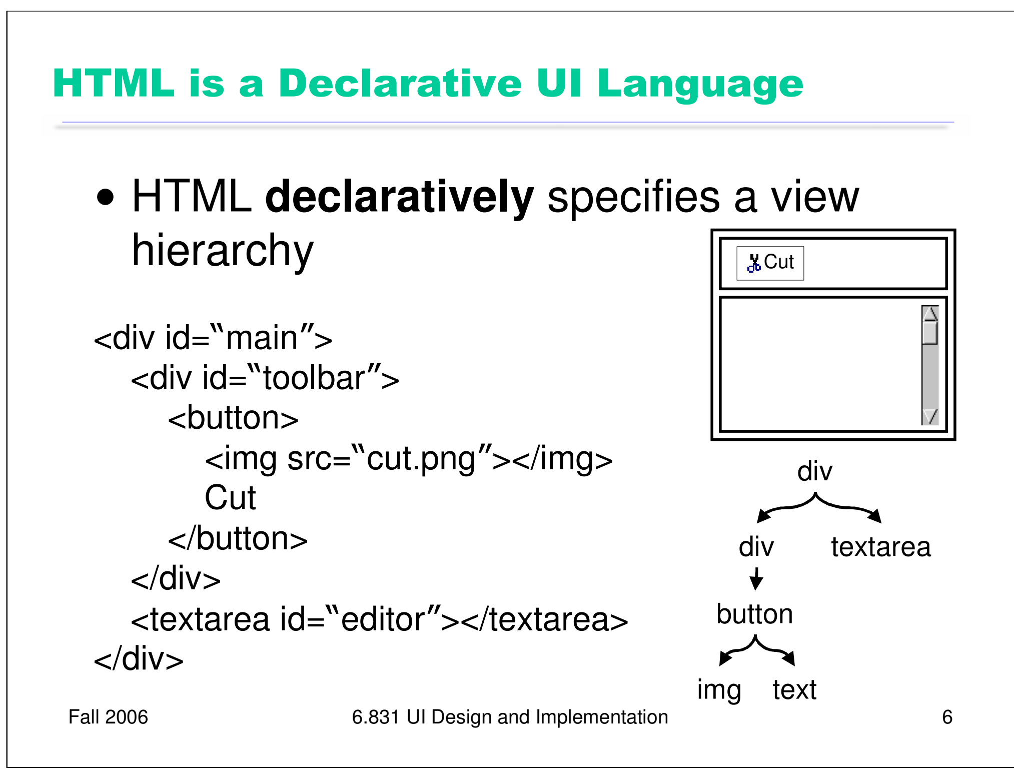
Declarative vs. Procedural

- Declarative programming
  - Saying *what* you want
- Procedural programming
  - Saying *how* to achieve it

Declarative
A tower of 3 blocks.

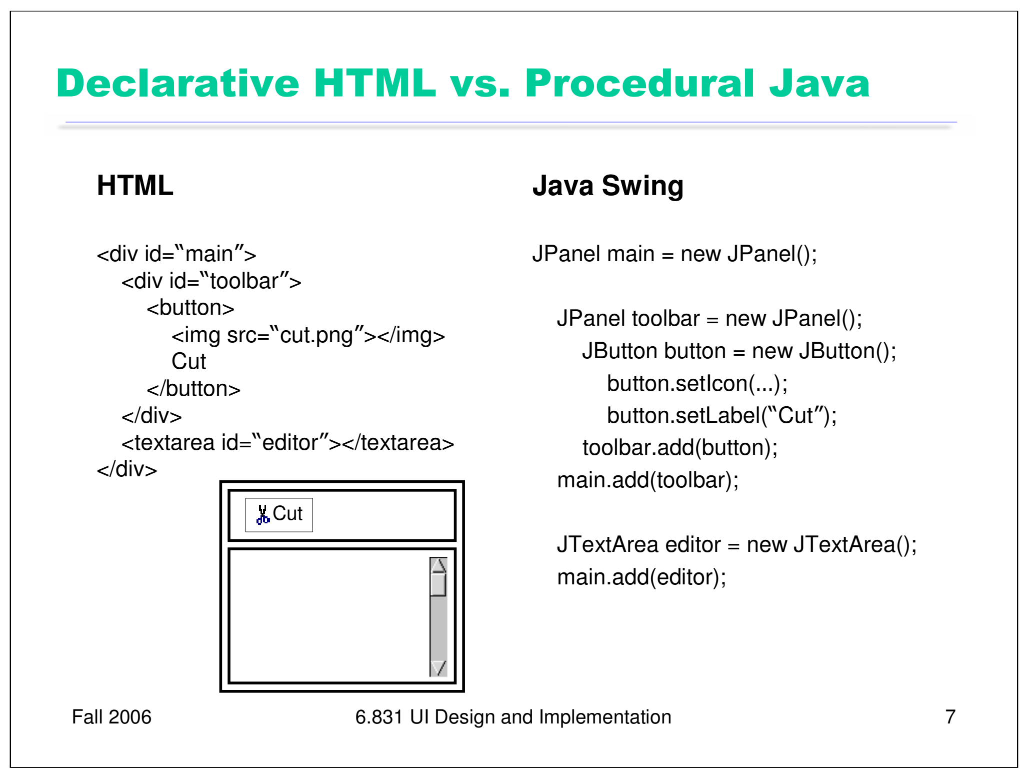
Procedural
1. Put down block A.
2. Put block B on block A.
3. Put block C on block B.

Today we'll be talking about ways to implement user interfaces using higher-level, more abstract specifications – particularly, **declarative specifications**. The key advantage of declarative programming is that you just say **what** you want, and leave it to an automatic tool to figure out how to produce it. That contrasts with conventional procedural programming, where the programmer has to say, step-by-step, how to reach the desired state.



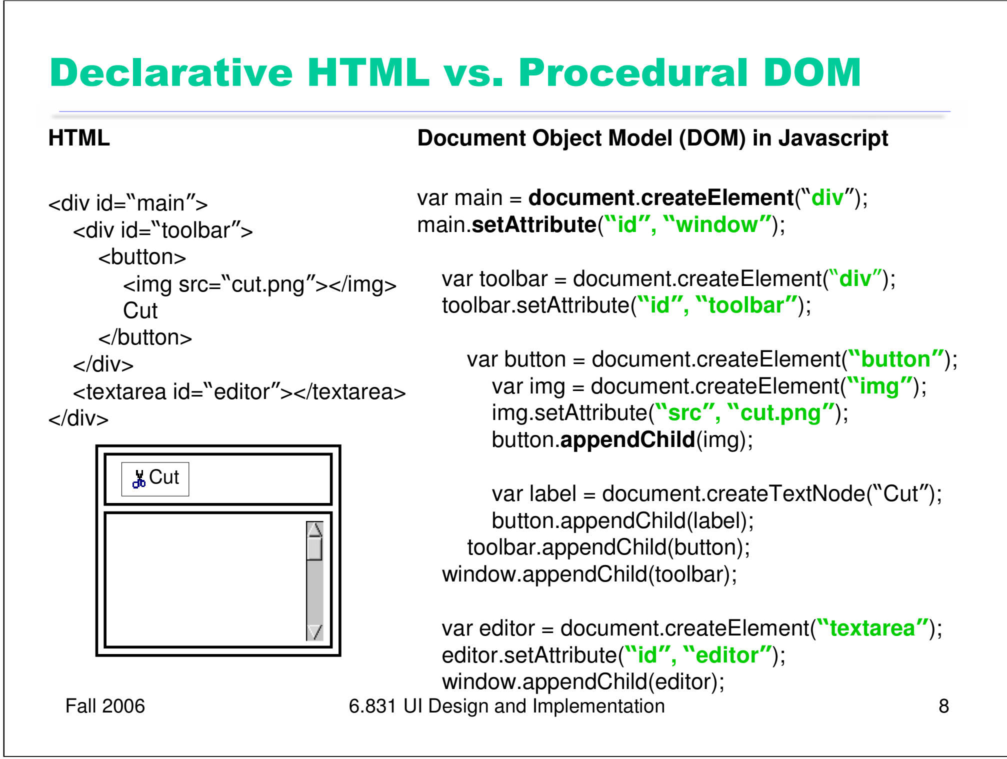
Our first example of declarative UI programming is **HTML**, which is a declarative specification of a view hierarchy. An HTML **element** is a component in the view hierarchy. The type of an element is its **tag**, such as div, button, and img. The properties of an element are its **attributes**. In the example here, you can see the id attribute (which gives a unique name to an element) and the src attribute (which gives the URL of an image to load in an img element); there are of course many others.

There's an automatic algorithm, built into every web browser, that constructs the view hierarchy from an HTML specification – it's simply an HTML parser, which matches up start tags with end tags, determines which elements are children of other elements, and constructs a tree of element objects as a result. So, in this case, the automatic algorithm for this declarative specification is pretty simple. We'll see more complex examples later in the lecture.



To give an analogy that you should be familiar with, here's some Swing code that produces the same interface procedurally. By comparison, the HTML is more concise, more compact – a common advantage of declarative specification.

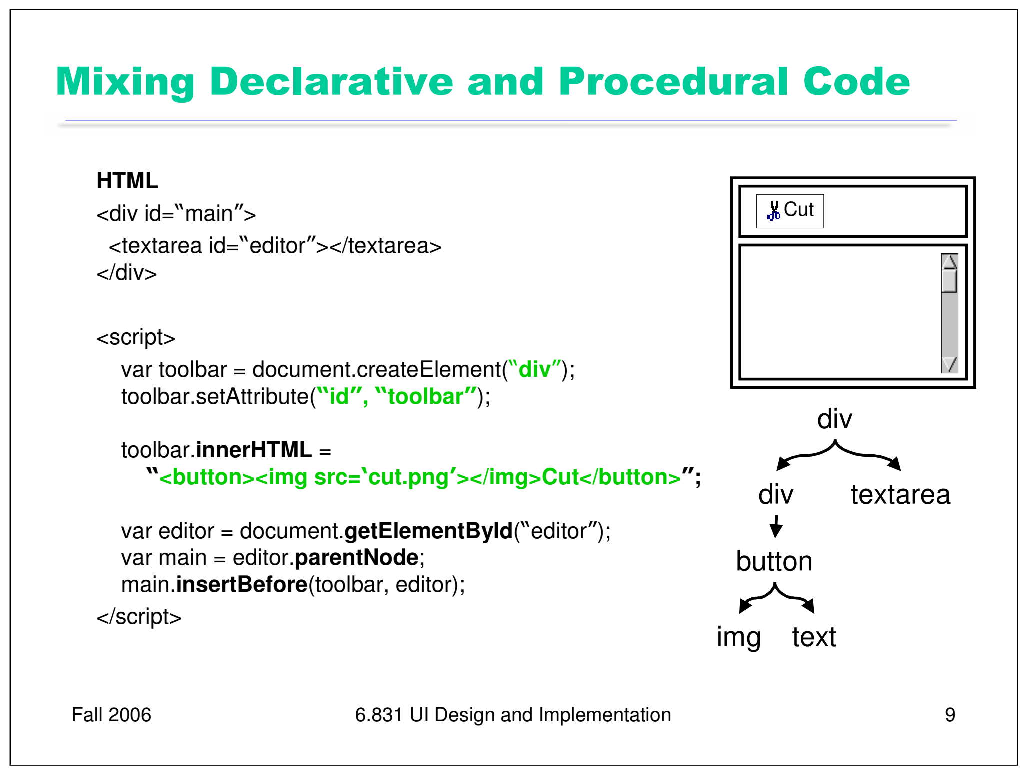
Note that neither the HTML nor the Swing code actually produces the **layout** shown in the picture, at least not yet. We'd have to add more information to both of them to get the components to appear with the right positions and sizes. We'll talk about layout later.



Here's procedural code that generates the same HTML component hierarchy, using the Javascript programming and the Document Object Model (DOM). DOM is a standard set of classes and methods for interacting with a tree of HTML or XML objects procedurally. (DOM interfaces exist not just in Javascript, which is the most common place to see it, but also in Java and other languages.)

There are a lot of similarities between the procedural code here and the procedural Swing code on the previous page – e.g. **createElement** is analogous to a constructor, **setAttribute** sets attributes on elements, and **appendChild** is analogous to add.

Incidentally, you don't always have to use the setAttribute method to change attributes on HTML elements. In Javascript, many attributes are reflected as properties of the element (analogous to fields in Java). For example, **obj.setAttribute("id", value)** could also be written as **obj.id = value**. Be warned, however, that only standard HTML attributes are reflected as object properties (if you call setAttribute with your own wacky attribute name, it won't appear as a Javascript property), and sometimes the name of the attribute is different from the name of the property. For example, the "class" attribute must be written as obj.className when used as a property.



To actually create a working interface, you frequently need to use a mix of declarative and procedural code. The declarative code is generally used to create the static parts of the interface, while the procedural code changes it dynamically in response to user input or model changes. Here's a (rather contrived) example that builds part of the interface declaratively, then fills it in with procedural code.

The **<script>** element allows you to introduce procedural code (which most web browsers assume is written Javascript) into the declarative specification. Code in the <script> element is executed immediately when the HTML page is first displayed, but of course you could also write functions or event handlers in the <script> element so that the procedural code runs later.

Even inside the procedural code, we can use declarative code. The **innerHTML** property of an HTML element represents the HTML between its start tag and end tag – in other words, the element's descendents in the view hierarchy. Setting this property removes all its current descendents and replaces them with elements created by the HTML you provide. Here, the button and img elements are created and added to the toolbar in this way.

The last part of the script shows a few other useful things. Putting **id** attributes on an element makes it easy to get a reference to it using **getElementById** in procedural code. (You can also refer to elements by id in declarative code.) You can also navigate around the element tree using **parentNode** and **childNodes[]** attributes. Also, you can insert new elements using **insertBefore**, not just append them; and you can remove and replace elements with **removeChild** and **replaceChild**. Documentation for these DOM operations can be found in many places on the Web; see the problem set for some useful references.

Advantages & Disadvantages of Declarative UI

- Usually more compact
- Programmer only has to know how to say *what*, not *how*
  - Automatic algorithms are responsible for figuring out how
- May be harder to debug
  - Can't set breakpoints, single-step, print in a declarative specification
  - Debugging may be more trial-and-error
- Authoring tools are possible
  - Declarative spec can be loaded and saved by a tool; procedural specs generally can't

Fall 2006 6.831 UI Design and Implementation 10

Now that we've worked through our first simple example of declarative UI – HTML – let's consider some of the advantages and disadvantages.

First, the declarative code is usually more compact than procedural code that does the same thing. That's mainly because it's written at a higher level of abstraction: it says *what* should happen, rather than *how*.

But the higher level of abstraction can also make declarative code harder to debug. There's generally no notion of time, so you can't use techniques like breakpoints and print statements to understand what's going wrong. The automatic algorithm that translates the declarative code into working user interface may be complex and hard to control – i.e., small changes in the declarative specification may cause large changes in the output. Declarative specs need debugging tools that are customized for the specification, and that give insight into how the spec is being translated; without those tools, debugging becomes trial and error.

On the other hand, an advantage of declarative code is that it's much easier to build authoring tools for the code, like HTML editors or GUI builders, that allow the user interface to be constructed by direct manipulation rather than coding. It's much easier to load and save a declarative specification than a procedural specification. Some GUI builders ***do*** use procedural code as their file format – e.g., generating Java code and automatically inserting it into a class. Either the code generation is purely one-way (i.e., the GUI builder spits it out but can't read it back in again), or the procedural code is so highly stylized that it amounts to a declarative specification that just happens to use Java syntax. If the programmer edits the code, however, they may deviate from the stylization and break the GUI builder's ability to read it back in.

Important HTML Elements for UI Design

- **Layout**
  - Box <div>
  - Grid <table>
- **Text**
  - Font & color <span>
  - Paragraph <p>
  - List <ul>,
- **Widgets**
  - Hyperlink <a>
  - Textbox <input type="text">
  - Textarea <textarea>
  - Drop-down <select>
  - Listbox <select multiple="true">
  - Button <input type="button">,<button>
  - Checkbox <input type="checkbox">
  - Radiobutton <input type="radio">
- **Pixel output**
  - <img>
- **Stroke output**
  - <canvas> (Firefox, Safari)
- **Procedural code**
  - <script>
- **Style sheets**
  - <style>

Fall 2006 6.831 UI Design and Implementation 11

To complete our survey of HTML as a language for generating component hierarchies, here is a cheat sheet of the most important elements that you might use in an HTML-based user interface.

The **<div>** and **<span>** elements are particularly important, and may be less familiar to people who have only used HTML for writing textual web pages. By default, these elements have no presentation associated with them; you have to add it using style rules (which we'll explain next). The <div> element creates a box (not unlike JPanel in Swing), and the <span> element changes textual properties like font and color while allowing its contents to flow and word-wrap.

HTML has a rather limited set of widgets. There are other declarative UI languages similar to HTML that have much richer sets of built-in components, such as XUL (used in Mozilla Firefox) and XAML (used in Microsoft Windows Vista).

HTML does support both pixel and stroke output, although the stroke output is nonstandard – some browsers support the <canvas> element, which has methods for making stroke output using procedural code, not much different from Swing's Graphics object.

Finally, we've already seen how to use the <script> element to embed procedural code (usually Javascript) into an HTML specification. The <style> element is used for embedding another declarative specification, style sheets, which is what we'll look at next.

Cascading Style Sheets (CSS)

- Key idea: separate the **structure** of the UI (view hierarchy) 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 separate style sheet defining pattern/style rules, possibly 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 isn't very interesting, because we've seen it before in Swing: changing styles directly on individual components. 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 to us here, 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.

CSS selectors aren't the only way to declaratively specify a set of HTML nodes (although it's the only way that's permitted in a CSS style sheet rule). Another declarative way to describe a set of elements is **XPath**, a pattern language that has some similarities to CSS selectors but is strictly more powerful.

Cascading and Inheritance

- If multiple rules apply to the same element, rules are automatically combined with **cascading** precedence
  - Source: browser defaults < web page < user overrides
    - *Browser says:* a { text-decoration: underline; }
    - *Web page says:* a { text-decoration: none; }
    - *User says:* a { text-decoration: underline; }
  - Rule specificity: general selectors < specific selectors
    - button { font-size: 12pt; }
    - .toolbarButton { font-size: 14pt; }
- Styles can also be **inherited** from element's parent
  - This is the default for simple styles like font, color, and text properties
    - body { font-size: 12pt; }

Fall 2006 6.831 UI Design and Implementation 14

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

```
<div id="main">
   <div id="toolbar">
      <button style="font-size: 12pt">
         <img src="cut.png"></img>
      </button>
   </div>
   <textarea id="editor"></textarea>
</div>
```

CSS

```
button { border: 2px; }
```

Javascript

```
var buttons =
     document.getElementsByTagName("button");
for (var i = 0; i < buttons.length; ++i) {
   var button = buttons[i];
   button.style.border = "2px";
   // not button.setAttribute("style", "border: 2px");
}
```

Fall 2006 6.831 UI Design and Implementation 15

Just as with HTML, we can change CSS styles procedurally as well. We saw earlier that HTML attributes can be get and set using Javascript object properties (like obj.id) rather than methods (like obj.setAttribute("id",…). For CSS styles, this technique is actually *essential*, since calling setAttribute() will replace the current style attribute entirely. In this example, if we called button.setAttribute("style", "border:2px"), the original style attribute (which set the font size to 12pt) would be lost. So it's best to use the style property, not the style attribute, when you're changing styles procedurally. The style property points to an object with properties representing all the style characteristics in CSS.

Automatic Layout

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

Our first example of declarative user interface should already be somewhat familiar to you: automatic layout. In Java, 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 component hierarchy. Then you specify how they should be related to each other, by attaching a layout manager to each container.

You can contrast this to a procedural approach to layout, in which you actually write Java or Javascript code that computes positions and sizes of graphical objects. You wrote a lot of this code in the checkerboard assignment, for example.

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 components

Fall 2006 6.831 UI Design and Implementation 17

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 generally laid out using absolute coordinates in a GUI builder, refuse to be resized!)

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 components -- 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.

Layout Manager Approach

- Layout manager performs automatic layout of a container's children
  - 1D (BoxLayout, FlowLayout, BorderLayout)
  - 2D (GridLayout, GridBagLayout, TableLayout)
- Advantages
  - Captures most common kinds of layout relationships in reusable, declarative form
- Disadvantages
  - Can only relate **siblings** in component hierarchy

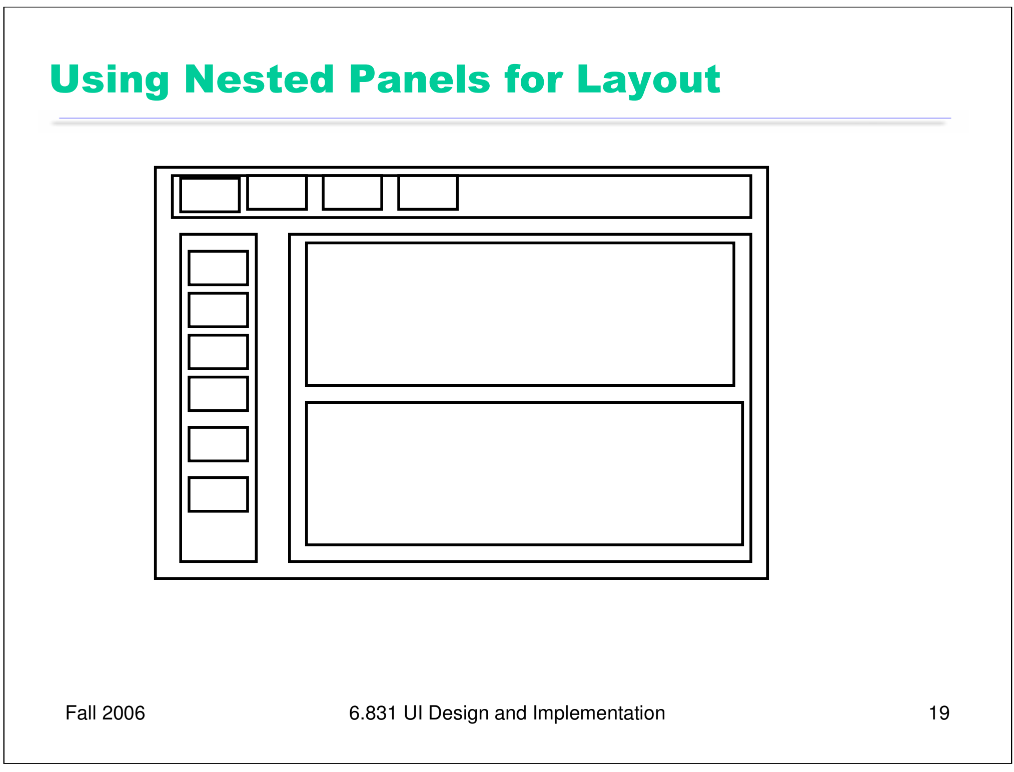
Fall 2006 6.831 UI Design and Implementation 18

Let's talk specifically about the layout-manager approach used in Java, which evolved from earlier UI toolkits like Motif and Tcl/Tk. A **layout manager** is attached to a container, and it computes the positions and sizes of that container's children. There are two basic kinds of layout managers: one-dimensional and two-dimensional.

One-dimensional layouts enforce only one direction of alignment between the components; for example, BoxLayout aligns components along a line either horizontally or vertically. BorderLayout is also one-dimensional: it can align components along any edge of the container, but the components on different edges aren't aligned with each other at all.

Two-dimensional layouts can enforce alignment in two directions, so that components are lined up in rows and columns. 2D layouts are generally more complicated to specify (totally GridBag!), but we'll see in the Graphic Design lecture that they're really essential for many dialog box layouts, in which you want to align captions and fields both horizontally and vertically at the same time.

Layout managers are a great tool because they capture the most common kinds of layout relationships as reusable objects. But a single layout manager can make only **local decisions**: that is, it computes the layout of **only one** container's children, based on the space available to the container. So they can only enforce relationships between **siblings** in the component hierarchy. For example, if you want all the buttons in your layout to be the same size, a layout manager can only enforce that if the buttons all belong to the same parent. That's a difference from the more general constraint system approach to layout that we'll see later in this lecture. Constraints can be global, cutting across the component hierarchy to relate different components at different levels.



Another common trick in layout is to introduce new containers (divs in HTML, JPanels in Java) in the component hierarchy, just for the sake of layout. This makes it possible to use one-dimensional layout managers more heavily in your layout. Suppose this example is Swing. A BorderLayout might be used at the top level to arrange the three topmost panels (toolbar at top, palette along the left side, and main panel in the center), with BoxLayouts to layout each of those panels in the appropriate direction.

This doesn't eliminate the need for two-dimensional layout managers, of course. Because a layout manager can only relate one container's children, you wouldn't be able enforce simultaneous alignments between captions and fields, for example, because using nested panels with one-dimensional layouts would force you to put them into separate containers.

Basic Layout Propagation Algorithm

```
computePreferredSize(Container parent)
    for each child in parent,
        computePreferredSize(child)
    compute parent's preferred size from children
        e.g., horizontal layout,
        (prefwidth,prefheight) = (sum(children prefwidth),
                                  max(children prefheight)

layout(Container parent)  requires: parent's size already set
    apply layout constraints to allocate space for each child
      child.(width,height) = (parent.width / #children, parent.height)
    set positions of children
      child[i].(x,y) = (child[i-1].x+child[i-1].width, 0)
    for each child in parent,
        layout(child)
```

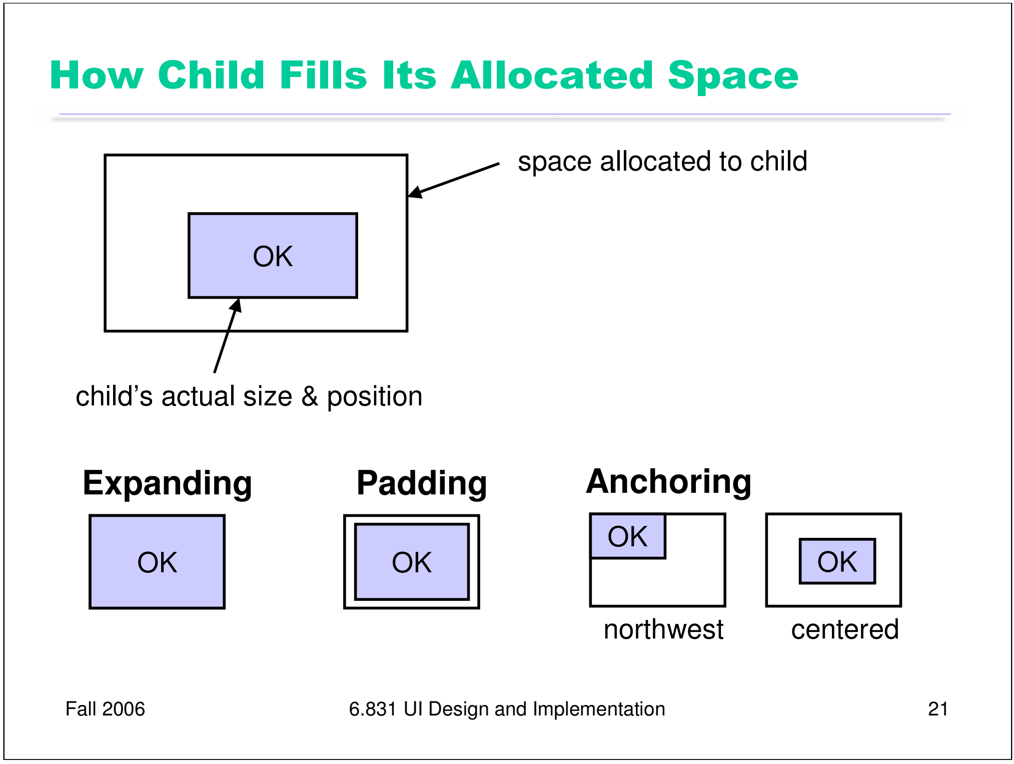
Fall 2006 6.831 UI Design and Implementation 20

Since the component hierarchy usually has multiple layout managers in it (one for each container), these managers interact by a **layout propagation** algorithm to determine the overall layout of the hierarchy.

Layout propagation has two parts.

First, the size requirements (**preferred sizes**) of each container are calculated by a **bottom-up** pass over the component hierarchy. The leaves of the hierarchy – like labels, buttons, and textboxes – determine their preferred sizes first, by calculating how large a rectangle they need to display to display their text label and surrounding whitespace or decorations. Then each container's layout manager computes its size requirement by combining the desired sizes of its children. The preferred sizes of components are used for two things: (1) to determine an initial size for the entire window, which is what Java's pack() method does; and (2) to allow some components to be fixed to their natural size, rather than trying to expand them or shrink them, and adjust other parts of the layout accordingly.

Once the size of the entire window has been established (either by computing its preferred size, or when the user manually sets it by resizing), the actual layout process occurs **top-down**. For each container in the hierarchy, the layout manager takes the container's assigned size (as dictated by its own parent's layout manager), applies the layout rules to allocate space for each child, and sets the positions and sizes of the children appropriately. Then it recursively tells each child to compute its layout.

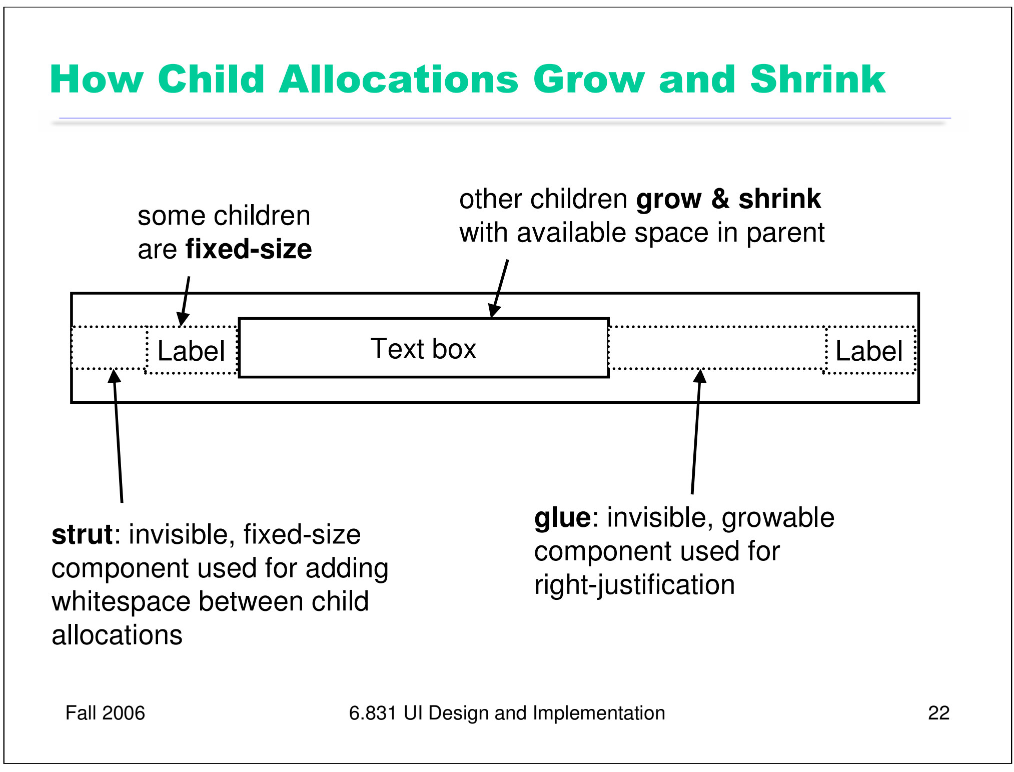


Let's talk about a few key concepts in layout managers. First, depending on the layout manager, the **space allocated** to a child by its container's layout manager is not always the same as the **size of the child**. For example, in GridBagLayout, you have to explicitly say that a component should **fill** its space allocation, in either the x or y direction or both (also called **expanding** in other layout managers).

Some layout managers allow some of the space allocation to be used for a margin around the component, which is usually called **padding**. The margin is added to the child's preferred size during the bottom-up size requirements pass, but then subtracted from the available space allocation during the top-down layout pass.

When a child doesn't fill its allocated space, most layout managers let you decide how you want the component to be **anchored** (or aligned) in the space – along a boundary, in a corner, or centering in one or both directions. In a sense, expanding is just anchoring to all four corners of the available space.

Since the boundaries aren't always visible – the button shown here has a clear border around it, but text labels usually don't – you might find this distinction between the space allocation and the component confusing. For example, suppose you want to left-justify a text label within the allocated space. You can do it two ways: (1) by telling the label itself to display left-justified with respect to its own rectangle, or (2) by telling the layout manager to anchor the label to the left side of its space allocation. But method #1 works only if the label is expanded to fill its space allocation, and method #2 works only if the label is **not** expanded. So subtle bugs can result.



Now let's look at how space allocations typically interact. During the top-down phase of the layout process, the container's size is passed down from above, so the layout manager has to do the best it can with the space provided to it. This space may be larger or smaller than the layout's preferred size. So layout managers usually let you specify which of the children are allowed to grow or shrink in response, and which should be fixed at their preferred size. If more than one child is allowed to take up the slack, the layout manager has rules (either built in or user-specified) for what fraction of the excess space should be given to each resizable child.

In Java, growing and shrinking is constrained by two other properties of components: minimum size and maximum size. So one way to keep a component from growing or shrinking is to ensure that its minimum size and maximum size are always identical to its preferred size. But layout managers often have a way to specify it explicitly, as well.

Struts and glue are two handy idioms for inserting whitespace (empty space) into an automatic layout. A **strut** is a fixed-size invisible component; it's used for margins and gaps between components. **Glue** is an invisible component that can grow and shrink with available space. It's often used to push components over to the right (or bottom) of a layout.

Sometimes the layout manager itself allows you to specify the whitespace directly in its rules, making struts and glue unnecessary. For example, TableLayout lets you have empty rows or columns of fixed or varying size. But BoxLayout doesn't, so you have to use struts and glue.

Java has factory methods for struts and glue in the Box class, but even if struts or glue weren't available in the toolkit, you could create them easily. Just make a component that draws nothing and set its sizes (minimum, preferred, maximum) appropriately.

HTML and CSS Layout

- Left-to-right, wrapping flow is the default
  Words in a paragraph (like Swing's FlowLayout)
- Absolute positioning in parent coordinate system

```
#B {
    position: absolute;
    left: 20px;
    width: 50%;
    bottom: 5em;
}
```


- 2D table layout
  <table>, <tr>, <td> (like ClearThought's TableLayout)

CSS layout offers three main layout strategies. The first is the default, left-to-right, wrapping flow typical of web pages. This is what Swing's FlowLayout also does.

More useful for UI design is **absolute** positioning, which allows a component's coordinates to specified either explicitly (in pixels relative to the parent's coordinate system) or relatively (as percentages of the parent). For example, setting a component's left to 50% would put it halfway across its parent's bounding box. Absolute positioning can constrain any two of the coordinates of a component: left, right, and width. (If it specifies all three, then CSS ignores one of them.)

Finally, HTML offers a table layout, which is flexible enough to handle most 2D alignments you'd want. The easiest way to use it is to use the <table> element and its family of related elements (<tr> for rows, and <td> for cells within a row).

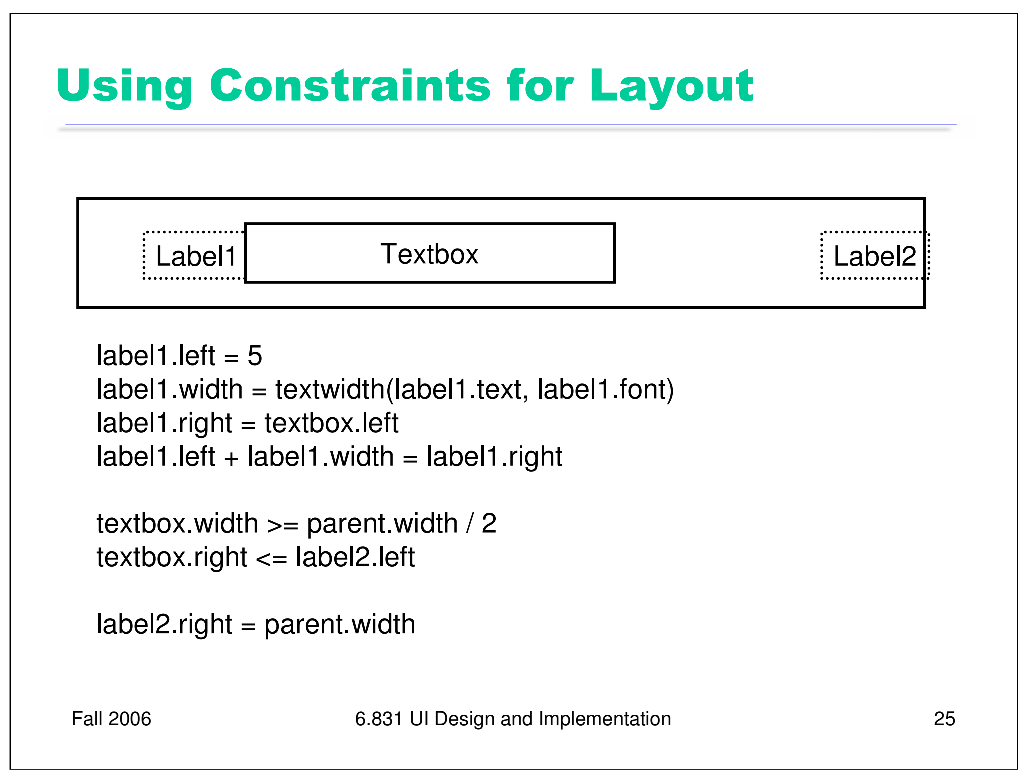
Constraints

- **Constraint** is a relationship among variables that is automatically maintained by system
  - **Constraint propagation:** When a variable changes, other variables are automatically changed to satisfy constraint

Since layout managers have 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**.

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 glue 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

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

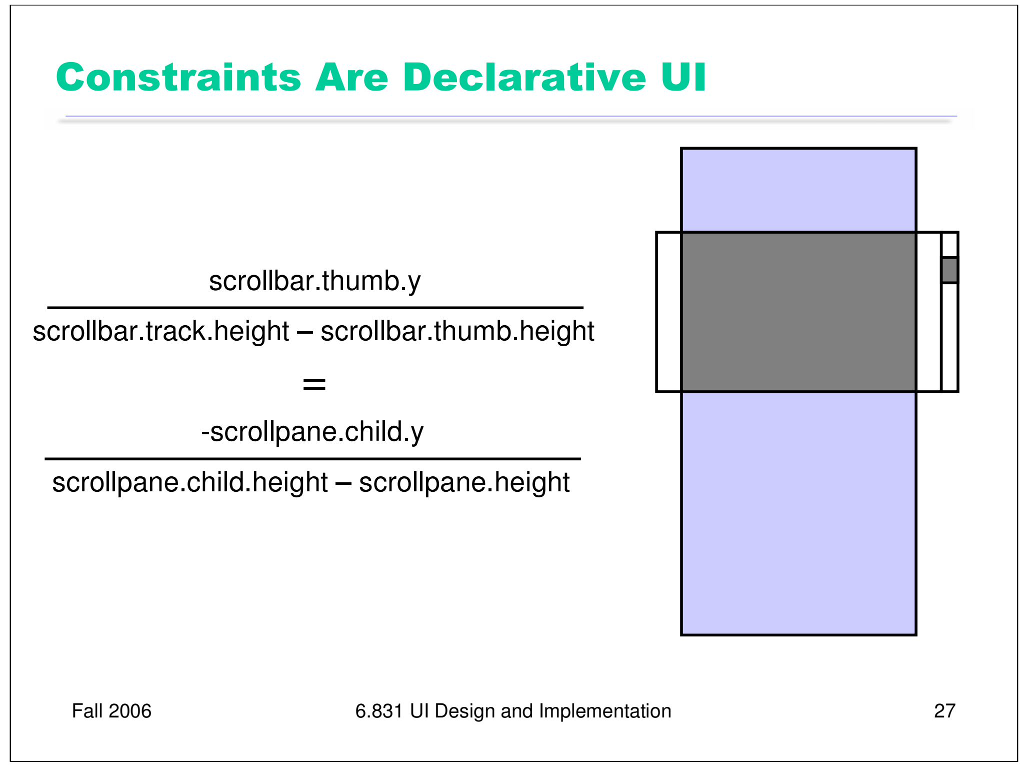
Constraints can be used for more general purposes than just layout. Here are a few.

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.



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 see how moving the thumb should affect the pane, and how moving the pane (e.g. by scrolling with arrow keys or jumping to a bookmark) should affect the thumb, so that 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.