Lecture 11: Output
Today’s hall of fame or shame candidate is the Domino’s Pizza build-your-own-pizza process. You can try it yourself by going to the Domino’s website and clicking Order to start an order (you’ll have to fill in an address to get to the part we care about, the pizza-building UI).

Some aspects to think about:
- learnability
- visibility
- user control & freedom
- efficiency
Today’s lecture continues our look into the mechanics of implementing user interfaces, by considering output in more detail.

One goal for these implementation lectures is not to teach any one particular GUI system or toolkit, but to give a survey of the issues involved in GUI programming and the range of solutions adopted by various systems. Presumably you’ve already encountered at least one GUI toolkit, probably Java Swing or HTML/Javascript. These lectures should give you a sense for what’s common and what’s unusual in the toolkit you already know, and what you might expect to find when you pick up another GUI toolkit.
There are basically three ways to represent the output of a graphical user interface.

**Components** is the same as the view hierarchy we discussed last week. Parts of the display are represented by view objects arranged in a spatial hierarchy, with automatic redraw propagating down the hierarchy. There have been many names for this idea over the years; the GUI community hasn’t managed to settle on a single preferred term.

**Strokes** draws output by making calls to high-level drawing primitives, like `drawLine`, `drawRectangle`, `drawArc`, and `drawText`.

**Pixels** regards the screen as an array of pixels and deals with the pixels directly.

All three output models appear in virtually every modern GUI application. The component model always appears at the very top level, for windows, and often for components within the windows as well. At some point, we reach the leaves of the view hierarchy, and the leaf views draw themselves with stroke calls. A graphics package then converts those strokes into pixels displayed on the screen. For performance reasons, a component may short-circuit the stroke package and draw pixels on the screen directly. On Windows, for example, video players do this using the DirectX interface to have direct control over a particular screen rectangle.

What model do each of the following representations use? HTML (component); Postscript laser printer (stroke input, pixel output); plotter (stroke input and output); PDF (stroke); LCD panel (pixel).
Since every application uses all three models, the design question becomes: at which points in your application do you want to step down into a lower-level output model? Here’s an example.

Suppose you want to build a view that displays a graph of nodes and edges. One approach would represent each node and each edge in the graph by a component (as in the tree on the right). Each node in turn might have two components, a circle and a text label. Eventually, you’ll get down to the primitive components available in your GUI toolkit. Most GUI toolkits provide a text label component; most don’t provide a primitive circle component. (One notable exception is SVG, which has component equivalents for all the common drawing primitives.) This would be a pure component model, at least from your application’s point of view – stroke output and pixel output would still happen, but inside primitive components that you took from the library.

Alternatively, the top-level window might have no subcomponents. Instead, it would draw the entire graph by a sequence of stroke calls: drawCircle for the node outlines, drawText for the labels, drawLine for the edges. This would be a pure stroke model.

Finally, your graph view might bypass stroke drawing and set pixels in the window directly. The text labels might be assembled by copying character images to the screen. This pure pixel model is rarely used nowadays, because it’s the most work for the programmer, but it used to be the only way to program graphics.

Hybrid models for the graph view are certainly possible, in which some parts of the output use one model, and others use another model. The graph view might use components for nodes, but draw the edges itself as strokes. It might draw all the lines itself, but use label components for the text.
Issues in Choosing Output Models

- Layout
- Input
- Redraw
- Drawing order
- Heavyweight objects
- Device dependence

Layout: Components remember where they were put, and draw themselves there. They also support automatic layout. With stroke or pixel models, you have to figure out (at drawing time) where each piece goes, and put it there.

Input: Components participate in event dispatch and propagation, and the system automatically does hit-testing (determining whether the mouse is over the component when an event occurs) for components, but not for strokes. If a graph node is a component, then it can receive its own click and drag events. If you stroked the node instead, then you have to write code to determine which node was clicked or dragged.

Redraw: An automatic redraw algorithm means that components redraw themselves automatically when they have to. Furthermore, the redraw algorithm is efficient: it only redraws components whose extents intersect the damaged region. The stroke or pixel model would have to do this test by hand. In practice, most stroked components don’t bother, simply redrawing everything whenever some part of the view needs to be redrawn.

Drawing order: It’s easy for a parent to draw before (underneath) or after (on top of) all of its children. But it’s not easy to interleave parent drawing with child drawing. So if you’re using a hybrid model, with some parts of your view represented as components and others as strokes, then the components and strokes generally fall in two separate layers, and you can’t have any complicated layering relationships between strokes and components.

Heavyweight objects: Every component must be an object (and even an object with no fields costs about 20 bytes in Java). As we’ve seen, the view hierarchy is overloaded not just with drawing functions but also with event dispatch, automatic redraw, and automatic layout, so that further bulks up the class. Views derived from large amounts of data – say, a 100,000-node graph – generally can’t use a component for every individual data item. The “flyweight” pattern can help, by storing redundant information in the component’s context (i.e., its parent) rather than in each component, but few toolkits support flyweight components. (See Glyphs: Flyweight Objects for User Interfaces by Paul R. Calder and Mark A. Linton. UIST ’90.)

Device dependence: The stroke model is largely device independent. In fact, it’s useful not just for displaying to screens, but also to printers, which have dramatically different resolution. The pixel model, on the other hand, is extremely device dependent. A directly-mapped pixel image won’t look the same on a screen with a different resolution.
As we said earlier, almost every GUI program uses all three models. At the highest level, a typical program presents itself in a window, which is a component. At the lowest level, the window appears on the screen as a rectangle of pixels. So a series of steps has to occur that translates that window component (and all the components it contains) into pixels.

The step from the component model down to the stroke model is usually called **drawing**. We’ll look at that first.

The step from strokes down to pixels is called **rasterization** (or scan conversion). The specific algorithms that rasterize various shapes are beyond the scope of this course (see 6.837 Computer Graphics instead). But we’ll talk about some of the effects of rasterization, and what you need to know as a UI programmer to control those effects.
Here’s how drawing works in the component model. Drawing is a top-down process: starting from the root of the component tree, each component draws itself, then draws each of its children recursively. The process is optimized by passing a clipping region to each component, indicating the area of the screen that needs to be drawn. Child components that do not intersect the clipping region are simply skipped, not drawn. In the example above, nodes B and C would not need to be drawn. When a component partially intersects the clipping region, it must be drawn – but any strokes or pixels it draws when the clipping region is in effect will be masked against the clip region, so that only pixels falling inside the region actually make it onto the screen.

For the root component, the clipping region might be the entire screen. As drawing descends the component tree, however, the clipping region is intersected with each component’s bounding box. So the clipping region for a component deep in the tree is the intersection of the bounding boxes of its ancestors.

For high performance, the clipping region is normally rectangular, using component bounding boxes rather than the components’ actual shape. But it doesn’t have to be that way. A clipping region can be an arbitrary shape on the screen. This can be very useful for visual effects: e.g., setting a string of text as your clipping region, and then painting an image through it like a stencil. Postscript was the first stroke model to allow this kind of nonrectangular clip region. Now many graphics toolkits support nonrectangular clip regions. For example, on Microsoft Windows and X Windows, you can create nonrectangular windows, which clip their children into a nonrectangular region.
Here’s an example of the redraw algorithm running on the graph window (starting with the clipping region shown on the last slide).

1. First the clip region is intersected with the whole window’s bounding box, and the window is told to draw itself within that intersection. The window draws its titlebar and its gray background. The window background effectively erases the previous contents of the window.

2. The window’s clip region is now intersected with its first child’s bounding box (Node A), and Node A is told to draw itself within that. In this particular example (where nodes are represented by circle and label components), Node A doesn’t do any of its own drawing; all the drawing will be handled by its children.

3. Now Node A’s circle child is told to draw itself. In this case, the circle has the same bounding box as Node A itself, so it receives the same clip region that Node A did. It draws a white circle.

4. Now Node A’s label child is told to draw itself, again using the same clip region because it has the same bounding box. It draws text on top of the circle just drawn.

5. Popping back up the tree, the next child of the window, Edge A-B, is told to draw itself, using the clip region that intersects its own bounding box with the window’s clip region. Only part of the edge falls in this clip region, so the edge only draws part of itself.

6. The next child of the window, Node B, doesn’t intersect the window’s clip region at all, so it isn’t told to draw itself.

7. The algorithm continues through the rest of the tree, either drawing children or skipping them depending on whether they intersect the clip region. (Would Edge A-C be drawn? Would Node C be drawn?)

Note that the initial clip region passed to the redraw algorithm will be different every time the algorithm is invoked. Clip regions generally come from *damage rectangles*, which will be explained in a moment.
When the bounding boxes of two objects overlap, like the circle and label components in the previous example, the redraw algorithm induces an ordering on the objects that makes them appear layered, one on top of the other. For this reason, 2D graphical user interfaces are sometimes called 2½D. They aren’t fully 3D, in which objects have x, y, and z coordinates; instead the z dimension is merely an ordering, called z order.

Z order is a side-effect of the order that the objects are drawn when the redraw algorithm passes over the hierarchy. Since drawing happens top-down, parents are generally drawn underneath children (although parents get control back after their children finish drawing, so a parent can draw some more on top of all its children if it wants). Older siblings (added to the view hierarchy earlier) are generally drawn underneath younger ones (added later). Java Swing is a curious exception to this – its redraw algorithm draws the highest-index child first, so the youngest sibling ends up on the bottom of the z order.

Z order can be affected by rearranging the tree, e.g. moving children to a different index position within their parent, or promoting them up the tree if necessary. This is often important for operations like drag-and-drop, since we generally want the object being dragged to appear on top of other objects.
When a component needs to change its appearance, it doesn’t repaint itself directly. It can’t, because the drawing process has to occur top-down through the component hierarchy: the component’s ancestors and older siblings need to have a chance to paint themselves underneath it. (So, in Java, even though a component can call its own paint() method directly, you generally shouldn’t do it!)

Instead, the component asks the graphics system to repaint it at some time in the future. This request includes a damaged region, which is the part of the screen that needs to be repainted. Often, this is just the entire bounding box of the component; but complex components might figure out which part of the screen corresponds to the part of the model that changed, so that only that part is damaged.

The repaint request is then queued for later. Multiple pending repaint requests from different components are consolidated into a single damaged region, which is often represented just as a rectangle – the bounding box of all the damaged regions requested by individual components. That means that undamaged screen area is being considered damaged, but there’s a tradeoff between the complexity of the damaged region representation and the cost of repainting.

Eventually – usually after the system has handled all the input events (mouse and keyboard) waiting on the queue -- the repaint request is finally satisfied, by setting the clipping region to the damaged region and redrawing the component tree from the root.
There’s an unfortunate side-effect of the automatic damage/redraw algorithm. If we draw a component tree directly to the screen, then moving a component can make the screen appear to flash – objects flickering while they move, and nearby objects flickering as well.

When an object moves, it needs to be erased from its original position and drawn in its new position. The erasure is done by redrawing all the objects in the view hierarchy that intersect this damaged region; typically the drawing of the window background is what does the actual erasure. If the drawing is done directly on the screen, this means that all the objects in the damaged region temporarily disappear, before being redrawn. Depending on how screen refreshes are timed with respect to the drawing, and how long it takes to draw a complicated object or multiple layers of the hierarchy, these partial redraws may be briefly visible on the monitor, causing a perceptible flicker.
Double-buffering solves this flickering problem. An identical copy of the screen contents is kept in a memory buffer. (In practice, this may be only the part of the screen belonging to some subtree of the view hierarchy that cares about double-buffering.) This memory buffer is used as the drawing surface for the automatic damage/redraw algorithm. After drawing is complete, the damaged region is just copied to screen as a block of pixels. Double-buffering reduces flickering for two reasons: first, because the pixel copy is generally faster than redrawing the view hierarchy, so there’s less chance that a screen refresh will catch it half-done; and second, because unmoving objects that happen to be caught, as innocent victims, in the damaged region are never erased from the screen, only from the memory buffer.

It’s a waste for every individual view to double-buffer itself. If any of your ancestors is double-buffered, then you’ll derive the benefit of it. So double-buffering is usually applied to top-level windows.

Why is it called double-buffering? Because it used to be implemented by two interchangeable buffers in video memory. While one buffer was showing, you’d draw the next frame of animation into the other buffer. Then you’d just tell the video hardware to switch which buffer it was showing, a very fast operation that required no copying and was done during the CRT’s vertical refresh interval so it produced no flicker at all.
In our description of the redraw algorithm, we said a component “draws itself,” meaning that it produces strokes to show itself on the screen. How that is actually done depends on the GUI toolkit you’re using.

In Java Swing (and many other desktop GUI toolkits, like Win32 and Cocoa), every component has a **drawing method**. In Swing, this method is `paint()`. The redraw algorithm operates by recursively calling `paint()` down the view hierarchy. Components can override the `paint()` method to change how they draw themselves. In fact, Swing breaks the `paint()` method down into several overridable template methods, like `paintComponent()` and `paintChildren()`, to make it easier to affect different parts of the redraw process. More about Swing’s painting process can be found in “Painting in AWT and Swing” by Amy Fowler (http://java.sun.com/products/jfc/tsc/articles/painting/).

In Adobe Flex (and the HTML `<canvas>` element supported by some browsers), there’s no drawing method available to override – the redraw algorithm is **hidden** from the programmer, much like the event loop is hidden by these toolkits. Instead, you make a sequence of stroke calls into the component’s graphics object, and the graphics object records this sequence of calls. Subsequently, whenever the component needs to redraw itself, it just plays back the recorded sequence of stroke calls. This approach is sometimes called **retained graphics**.

A key difference between these approaches is **when** stroke calls can be made. With the drawing method approach, drawing should only be done while the drawing method is active. Drawing done at a different time (like during an event handler) will not interact correctly with the redraw algorithm; it won’t respect z order, and it will be ephemeral, overwritten and destroyed the next time the redraw algorithm touches that component. With the retained graphics approach, however, the stroke calls can be recorded at any time, and the toolkit automatically handles playing them back at the right point in the redraw. The retained graphics approach tends to be less error prone for a programmer; drawing at the wrong time is a common mistake for beginning Swing programmers.

A potential downside of the retained graphics approach is performance. The recorded strokes must be stored in memory. Although this recording is not as heavyweight as a component tree (since it doesn’t have to handle input or layout, or even necessarily be represented as objects), you probably wouldn’t want to do it with millions of stroke calls. So if you had an enormous view (like a map) being displayed inside a scrolling pane (so that only a small part of it was visible on screen), you wouldn’t want to stroke the entire map. The drawing method approach gives more control over this; since you have access to the clip region in the drawing method, you can choose not to render strokes that would be clipped. To do the equivalent thing with retained graphics would put more burden on the programmer to determine the visible rectangle and rerecord the stroke calls every time this rectangle changed.
Now let’s look at the drawing capabilities provided by the stroke model.

Every toolkit’s stroke model has some notion of a **drawing surface**. The screen is only one possible place where drawing might go. Another common drawing surface is a memory buffer, which is an array of pixels just like the screen. Unlike the screen, however, a memory buffer can have arbitrary dimensions. The ability to draw to a memory buffer is essential for double-buffering. Another target is a printer driver, which forwards the drawing instructions on to a printer. Although most printers have a pixel model internally (when the ink actually hits the paper), the driver often uses a stroke model to communicate with the printer, for compact transmission. Postscript, for example, is a stroke model.

Most stroke models also include some kind of a **graphics context**, an object that bundles up drawing parameters like color, line properties (width, end cap, join style), fill properties (pattern), and font.

The stroke model may also provide a current **coordinate system**, which can be translated, scaled, and rotated around the drawing surface. We’ve already discussed the **clipping region**, which acts like a stencil for the drawing. Finally, a stroke model must provide a set of **drawing primitives**, function calls that actually produce graphical output.

Many systems combine all these responsibilities into a single object. Java’s Graphics object is a good example of this approach. In other toolkits, the drawing surface and graphics context are independent objects that are passed along with drawing calls.

When state like graphics context, coordinate system, and clipping region are embedded in the drawing surface, the surface must provide some way to save and restore the context. A key reason for this is so that parent views can pass the drawing surface down to a child’s draw method without fear that the child will change the graphics context. In Java, for example, the context can be saved by Graphics.create(), which makes a copy of the Graphics object. Notice that this only duplicates the graphics context; it doesn’t duplicate the drawing surface, which is still the same.
When you’re using a stroke model, it’s important to understand how the strokes are actually converted into pixels. Different platforms make different choices.

One question concerns how stroke coordinates, which represent zero-dimensional points, are translated into pixel coordinates, which are 2-dimensional squares. Microsoft Windows places the stroke coordinate at the center of the corresponding pixel; Java’s stroke model places the stroke coordinates between pixels.

The other questions concern which pixels are actually drawn when you request a line or a rectangle.
It’s beyond the scope of this lecture to talk about algorithms for converting a stroke into pixels. But you should be aware of some important techniques for making strokes look good.

One of these techniques is **antialiasing**, which is a way to make an edge look smoother. Instead of making a binary decision between whether to color a pixel black or white, antialiasing uses a shade of gray whose value varies depending on how much of the pixel is covered by the edge. In practice, the edge is between two arbitrary colors, not just black and white, so antialiasing chooses a point on the gradient between those two colors. The overall effect is a fuzzier but smoother edge.

Subpixel rendering takes this a step further. Every pixel on an LCD screen consists of three discrete pixels side-by-side: red, green, and blue. So we can get a horizontal resolution which is three times the nominal pixel resolution of the screen, simply by choosing the colors of the pixels along the edge so that the appropriate subpixels are light or dark. It only works on LCD screens, not CRTs, because CRT pixels are often arranged in triangles, and because CRTs are analog, so the blue in a single “pixel” usually consists of a bunch of blue phosphor dots interspersed with green and red phosphor dots. You also have to be careful to smooth out the edge to avoid color fringing effects on perfectly vertical edges. And it works best for high-contrast edges, like this edge between black and white. Subpixel rendering is ideal for text rendering, since text is usually small, high-contrast, and benefits the most from a boost in horizontal resolution. Windows XP includes ClearType, an implementation of subpixel rendering for Windows fonts. (For more about subpixel rendering, see Steve Gibson, “Sub-Pixel Font Rendering Technology”, http://grc.com/cleartype.htm)
Finally, let’s talk in more detail about what the pixel model looks like.

Put simply, it’s a rectangular array of pixels – but pixels themselves are not always so simple. A pixel itself has a depth, so this model is really three dimensional. Depth is often expressed in bits per pixel (bpp). The simplest kind of pixel model has 1 bit per pixel; this is suitable for representing black and white images. It’s also used for bitmasks, where the single-bit pixels are interpreted as boolean values (pixel present or pixel missing). Bitmasks are useful for clipping – you can think of a bitmask as a stencil.

Another kind of pixel representation uses each pixel value as an index into a palette, which is just a list of colors. In the 4-bpp model, for example, each of the 16 possible pixel values represents a different color. This kind of representation, often called Indexed Color, was useful when memory was scarce; you still see it in the GIF file format, but otherwise it isn’t used much today.

The most common pixel representation is often called “true color” or “direct color”; in this model, each pixel represents a color directly. The color value is usually split up into multiple components: red, green, and blue. (Color components are also called channels or bands; the red channel of an image, for example, is a rectangular array of the red values of its pixels.)

A pixel model can be arranged in memory (or a file) in various ways: packed tightly together to save memory, or spread out loosely for faster access; with color components interleaved or separated; and scanned from the top (so that the top-left pixel appears first) or the bottom (the bottom-left pixel appearing first).
Many pixel models have a fourth channel in addition to red, green, and blue: the pixel’s **alpha** value, which represents its degree of transparency. We’ll talk more about alpha in a future lecture.

<table>
<thead>
<tr>
<th>Transparency</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>Alpha</strong> is a pixel’s transparency</td>
</tr>
<tr>
<td>- from 0.0 (transparent) to 1.0 (opaque)</td>
</tr>
<tr>
<td>- so each pixel has red, green, blue, and alpha values</td>
</tr>
<tr>
<td>• Uses for alpha</td>
</tr>
<tr>
<td>- Antialiasing</td>
</tr>
<tr>
<td>- Nonrectangular images</td>
</tr>
<tr>
<td>- Translucent components</td>
</tr>
<tr>
<td>- Clipping regions with antialiased edges</td>
</tr>
</tbody>
</table>
The primary operation in the pixel model is copying a block of pixels from one place to another – often called **bitblt** (pronounced “bit blit”). This is used for drawing pictures and icons on the screen, for example. It’s also used for double-buffering – after the offscreen buffer is updated, its contents are transferred to the screen by a bitblt.

Bitblt is also used for screen-to-screen transfers. To do fast scrolling, for example, you can bitblt the part of the window that doesn’t change upwards or downwards, to save the cost of redrawing it. (For example, look at Swing’s JViewport.BLIT_SCROLL_MODE.)

It’s also used for sophisticated drawing effects. You can use bitblt to combine two images together, or to combine an image with a mask, in order to clip it or composite them together.

Bitblt isn’t always just a simple array copy operation that replaces destination pixels with source pixels. There are various different rules for combining the destination pixels with the source pixels. These rules are called **compositing** (**alpha compositing**), when the images have an alpha channel), and we’ll talk about them in a later lecture.
Here are a few common image file formats. It’s important to understand when to use each format. For user interface graphics, like icons, JPG generally should not be used, because it’s lossy compression – it doesn’t reproduce the original image exactly. When every pixel matters, as it does in an icon, you don’t want lossy compression. JPG also can’t represent transparent pixels, so a JPG image always appears rectangular in your interface.

For different reasons, GIF is increasingly unsuitable for interface graphics. GIF isn’t lossy – you get the same image back from the GIF file that you put into it – but its color space is very limited. GIF images use 8-bit color, which means that there can be at most 256 different colors in the image. That’s fine for some low-color icons, but not for graphics with gradients or blurs. GIF has limited support for transparency – pixels can either be opaque (alpha 1) or transparent (alpha 0), but not translucent (alpha between 0 and 1). So you can’t have fuzzy edges in a GIF file, that blend smoothly into the background. GIF files can also represent simple animations.

PNG is the best current format for interface graphics. It supports a variety of color depths, and can have a full alpha channel for transparency and translucency. (Unfortunately Internet Explorer 6 doesn’t correctly display transparent PNG images, so GIF still rules web graphics.) If you want to take a screenshot, PNG is the best format to store it.
Our final topic in today’s lecture concerns describing output 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 a markup language, such as HTML. A markup language provides 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 Javascript 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.

There are also toolkits that substantially simplify procedural programming in HTML – jQuery is a good example.
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 descendants 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[]` properties. 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 class wiki for some useful references.
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.
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 MXML (used in Adobe Flex) and XUL (used in Mozilla Firefox) and XAML (used in Microsoft WPF and Silverlight).

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.
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.
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

- 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; }

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.
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.
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.
This lecture has discussed the most widely-used declarative user interface language (HTML), but other up-and-coming languages include MXML, XUL, and XAML.

SVG is worth noting, too. SVG is a markup language for vector graphics, like lines, curves, shapes, etc. You might think of it as a component model that goes all the way down to primitive stroke shapes. Many drawing programs can export SVG files, and many web browsers (and Adobe Flex) can render SVG. SVG files may soon become a viable alternative to image files for user interface graphics; since they represent strokes, they are resolution independent and able to be transformed (resized, rotated, etc.) while maintaining high quality rendering.

All these XML-based languages support CSS for styling, so the investment you make in learning CSS will pay off across different toolkits.
A final word about debugging the output of a graphical user interface, which can sometimes be tricky. A common problem is that you try to draw something, but it never appears on the screen. Here are some possible reasons why.

Wrong place: what’s the origin of the coordinate system? What’s the scale? Where is the component located in its parent?

Wrong size: if a component has zero width and zero height, it will be completely invisible no matter what it tries to draw—everything will be clipped. Zero width and zero height are the defaults for all components in Swing—you have to use automatic layout or manual setting to make it a more reasonable size. Check whether the component (and its ancestors) have nonzero sizes.

Wrong color: is the drawing using the same color as the background? Is it using 100% alpha, so that it’s completely transparent?

Wrong z-order: is something else drawing on top?
Summary

- Component, stroke, pixel models
- Automatic redraw and double-buffering
- Image formats
- Declarative UI: markup languages and CSS