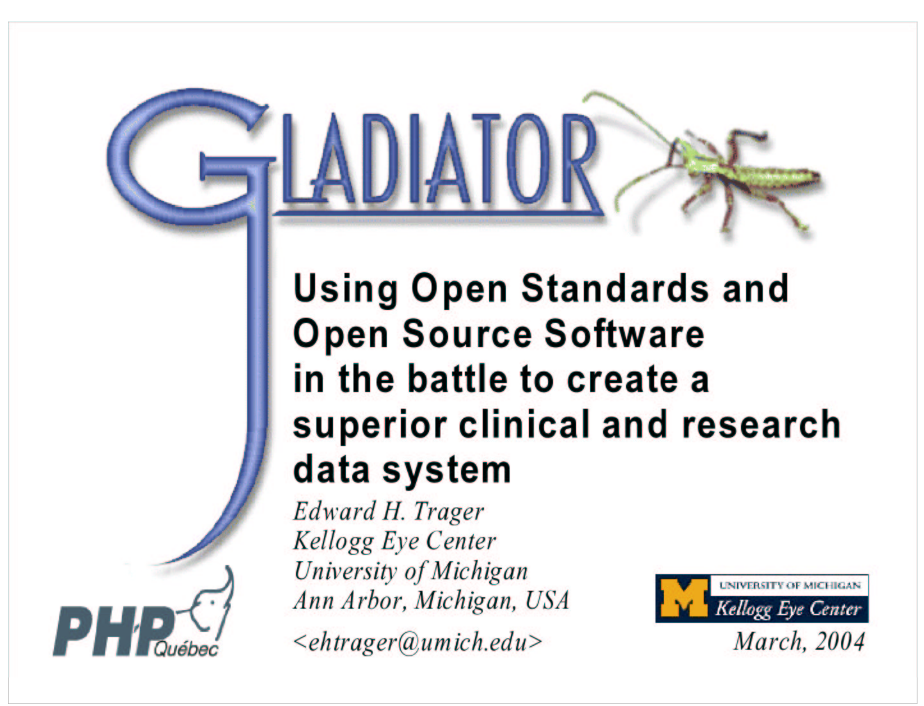


Gladiator: Using Open Standards and Open Source Software in the battle to create a superior clinical and research data system.

Clinicians

- Often stuck using legacy thick-client systems
- Systems were designed to meet clinicians needs
- No consideration was given to research needs

Researchers

- Usually using ad-hoc systems
- Little consideration given to future maintainability
- Often impossible to integrate research system with clinical system

In research hospitals, patients are often recruited to participate in research studies. In most cases, clinical data on these patients is entered into existing proprietary database systems that were designed to meet the needs of the clinicians alone.

The needs of the researchers who will analyze the collected data were often not considered when these systems were designed.

Based on proprietary technologies, the clinical systems usually use thick client architectures which impose a number of severe limitations.

Limitations of Existing Systems

- Difficult to modify existing clinical systems to meet the additional needs imposed by the research protocols.
- Difficult to achieve adequate security and legislated privacy of patient information when redeploying a clinically-centered database to provide researchers access to information they require.
- Usually limitations cannot be overcome, forcing clinical and research teams to find sub-optimal work-arounds to the systems.
- Working around limitations results in wasted time and effort and introduces uneccessary errors.

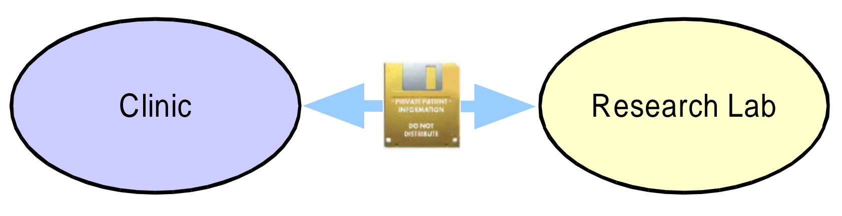
One class of limitations centers around trying to modify existing clinical systems to meet the additional needs imposed by the research protocols.

Another class of limitations centers around trying to achieve appropriate security and privacy of patient information when considering how to redeploy a clinically-centered system so that researchers can attain access to the information they need.

Usually the limitations cannot be overcome, forcing both clinical and research teams to work in sub-optimal ways that waste time and result in unecessary errors, delays, even skewed results.

Working Around Limitations

- Since the clinical systems are insufficient, research labs often create ad-hoc systems in house.
- Little thought given to future maintainability of the research database system as requirements change over time.
- There is often almost no integration between the clinicians and researchers' respective systems.



The research teams often need to develop database systems to better cope with the data extracted from the clinical systems.

In addition, researchers need to store additional data, such as genetic data obtained from blood samples in the lab.

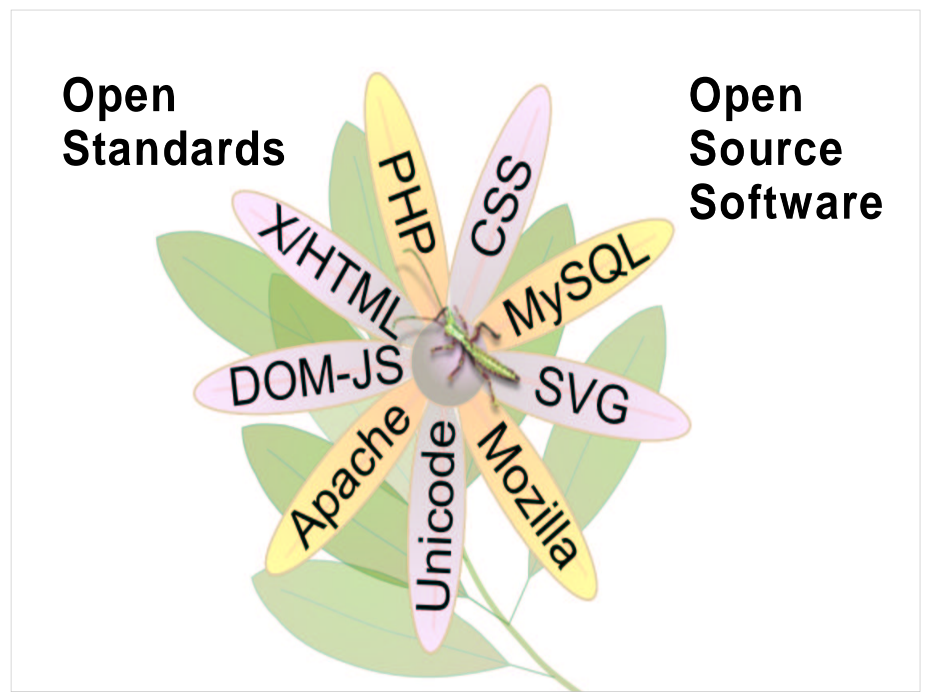
The research systems are often in-house designs with little thought given to future maintainability as requirements change over time.

The **Gladiator** project
is our attempt to thoughtfully
bridge the gap between
the clinic and the research lab
using **Open Standards**
and **Open Source** technologies

The **Gladiator** project is our attempt to thoughtfully bridge the gap between the clinic and the research lab by using **Open Standards** and **Open Source** technologies to build from scratch a secure and completely integrated system for the entry of clinical and genetic research data on eye diseases.

(Tomorrow (Friday) at 9:00 AM **Frank Boumphrey** of **Cormorant Software** will describe how he is using PHP to build a front end on top of proprietary hospital information systems and how PHP is allowing them to release data bound up in such systems. Mr. Boumphrey's approach is almost opposite to our approach. After listening to this talk today, it might be very interesting to attend Mr. Boumphrey's talk tomorrow.)

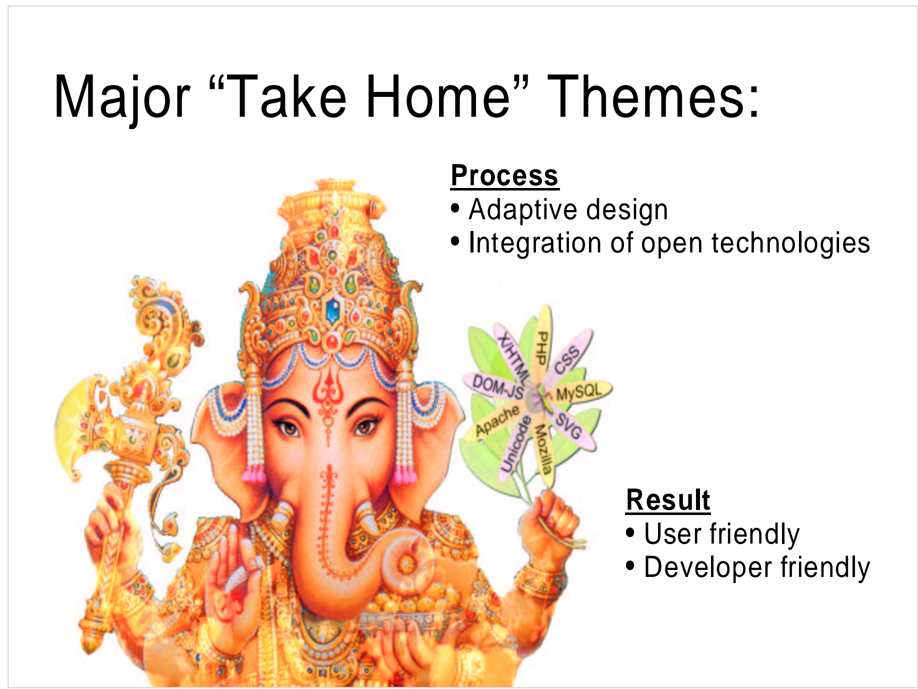
The **Gladiator** name comes from the name assigned to the new order of insects, the *Mantophasmatodea*, recently discovered by Oliver Zompro.



Our goal is to create a secure, integrated, standards-compliant, web-based data system for the entry of clinical and genetic research data used in the study of inherited eye diseases.

To achieve that goal, we are building Gladiator using **PHP**, **MySQL**, the **Apache** web server, and, on the client side, the **Mozilla** browser.

We are making extensive use of the features provided by **X/HTML**, **CSS**, **DOM**-based **ECMAScript** (**Javascript**), and **SVG** in order to keep our code clean, simple, modular, and easy to understand.



A major theme that I will stress throughout this talk today is that an adaptive design process combined with the integration of open technologies and open standards can result in a system that is both *user friendly* and *developer friendly*.

When you design a system, you want to ask yourself both of these questions:

- Is it user friendly?
- Is it developer friendly?

CSS and DOM Implementations by Browser

	Engine			
	Gecko	KHTML	Opera	IE
	Mozilla Firefox	Konqueror Safari	Opera	IE
CSS	☆☆☆☆☆	☆☆☆	☆☆☆☆	☆☆
DOM	☆☆☆☆☆	☆☆	☆☆	☆

** Versions: **Mozilla** 1.4, 1.5, 1.6, 1.7α; **Konqueror** 3.1.4 (3.2); **Safari** 1.0, (1.2); **Opera** 6.0, 7.11, 7.23; **Internet Explorer** 5.5, 6.0. Platforms: Linux, Mac OS X, Win2K.*

We have been testing our code on all four major browser engines: **Gecko**, **KDE**'s **KHTML**, **Opera**, and **Internet Explorer**.

Of these, **Mozilla** (**Gecko**) is the only browser able to handle every **CSS** and **DOM** feature we have “thrown” at it to date in **Gladiator**.

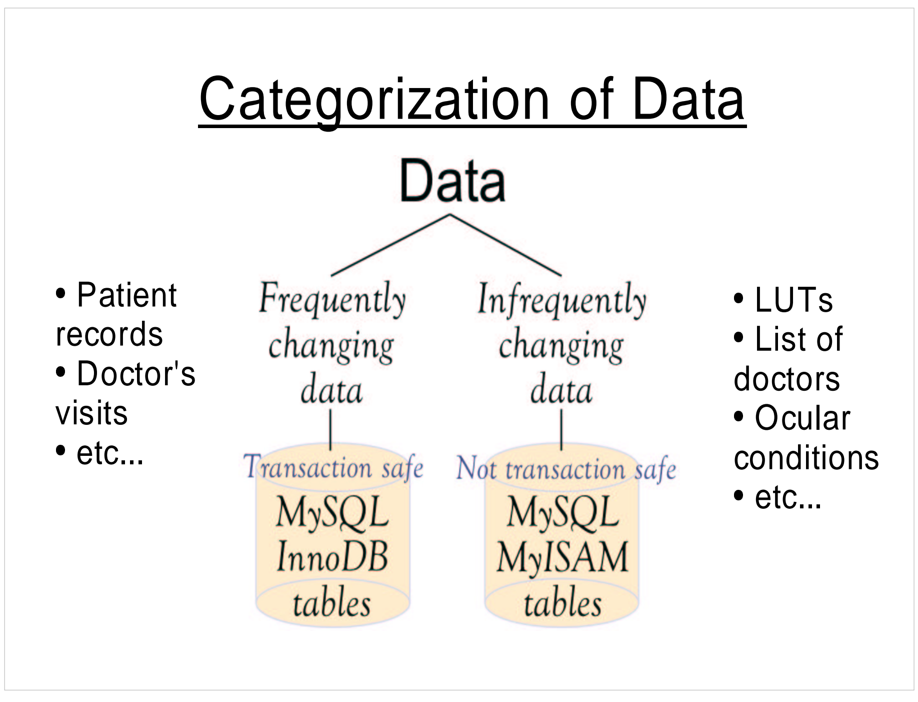
We continue to see great improvements in **CSS** and **DOM** handling by **Konqueror**, **Safari**, and **Opera** which, like **Mozilla**, are released frequently. However current limitations, especially in **DOM** handling, by these browsers means we can't certify them for **Gladiator** at the present time.

Given its market share, **Internet Explorer**'s poor handling of **CSS** and **DOM** is disappointing.

DATA

I would now like to talk about the data in our project at some length.

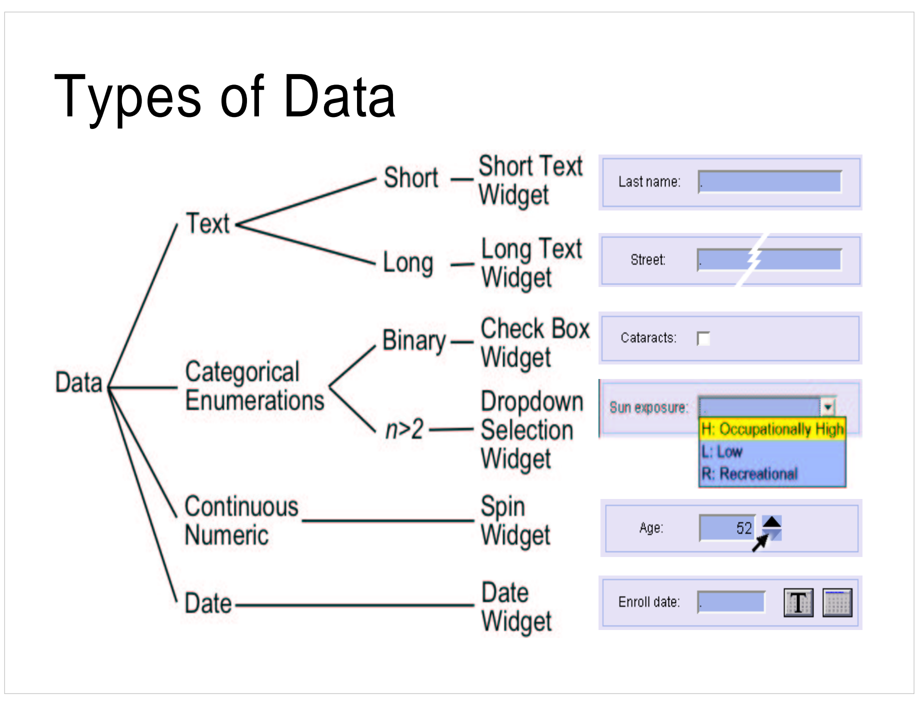
Although of course much of the data are specific to this project, the general approach that we have taken, and way that we have conceptually organized the data are completely applicable to other projects.



During the initial design phase of a large project like this, one of the first things you want to do is determine which classes of data in the system will change rapidly (*dynamic*), and which will remain *static* or nearly static over long periods of time.

When we started, we debated at length whether we should use **Oracle** or **MySQL**. Professors in the Computer Science and Electrical Engineering department at UM with whom we had partnered suggested we stick with **MySQL** because it is much simpler than **Oracle** to set up and tune. This was an important factor because we planned to have recent CS grads working on the project.

We are now taking advantage of transaction support provided by the **InnoDB** container provided in recent **MySQL** releases, while maintaining more static data in traditional **MyISAM** tables.



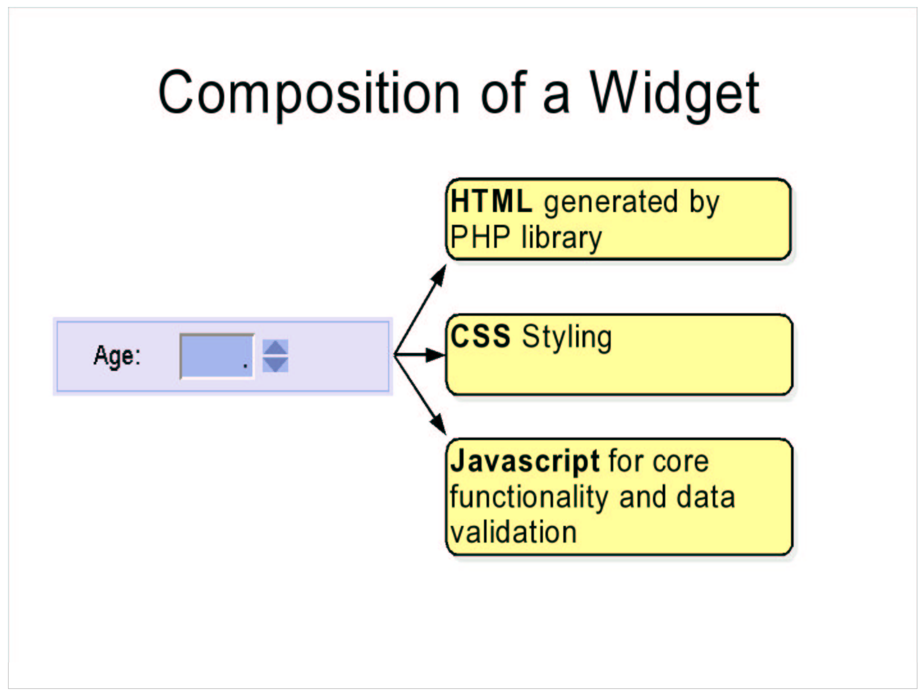
After gaining a rough idea of what tables we would need and whether those tables fell into the *dynamic* or *static* data classes, we took a look at the types of data we would be needing to enter into the system.

There are just four basic types: *1) text, 2) categorical enumerations, 3) continuous numeric*, and *4) date*.

Some of these basic types are broken down into sub-types, like *long text* vs. *short text,* and *binary* vs. *n>2 enumerations*.

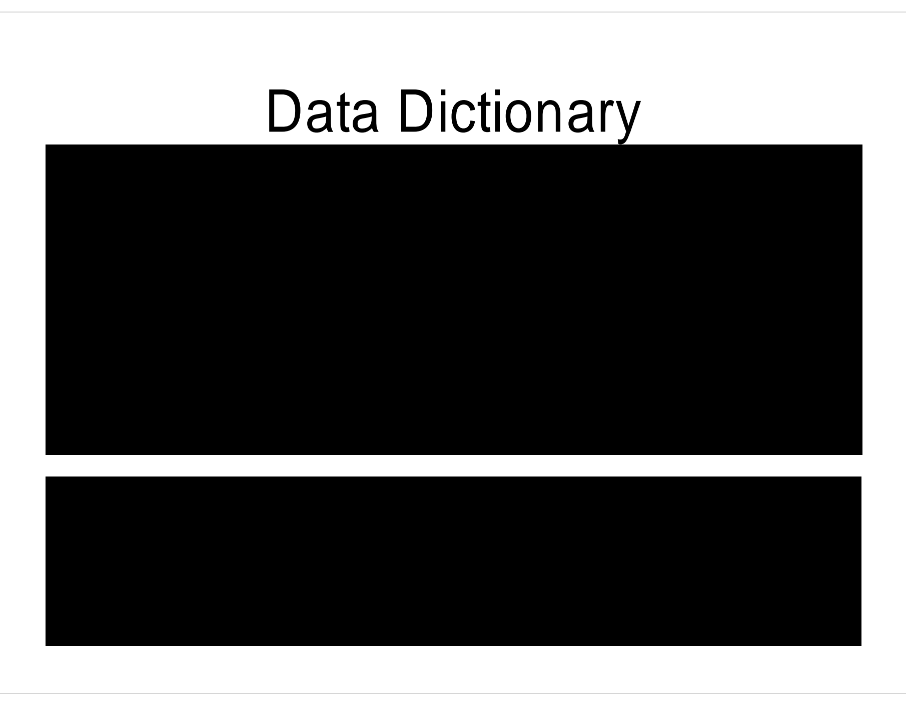
For a scientific system like ours, there are actually very few *binary enumerations*: many end up being *n>2 enumerations* because we need to have a *missing value indicator*, i.e. *gender{male,female, unknown}*, or *SunExposure{high,recreational,low,missing}*.

After determining the data types, we decided what the form interface elements (*widgets*) should be for each type of data.



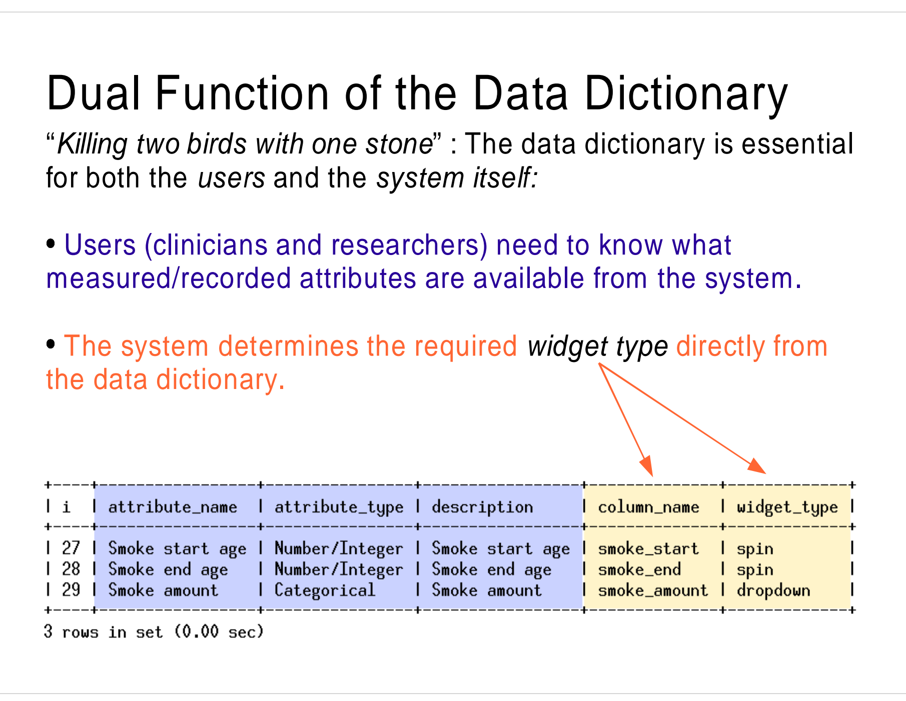
Each widget consists of:

(1) HTML generated from our PHP library
(2) associated CSS styling
(3) Javascript code needed for (a) core functionality and for (b) data validation.



Metadata about every single measured/recorded attribute is stored as record in a data dictionary.

In addition to the attribute *name*, *type*, and *description*, the table stores the *column_name* and *widget_type* required by the attribute.



The *data dictionary* system table serves two important purposes.

First, it is a list all of the attributes measured or recorded on the patients. The users (both clinicians and researchers) can browse or search this table to find the description of an attribute. For categorically-coded attributes (which appear as drop-down lists in Gladiator), they can also click on an attribute to find out the codes and a description of the codes used for that attribute.

Secondly, the layout manager uses this table to determine which widget to use to display a given attribute to the user in a form.

Layout Manager

```
//////////////////////////////////
//
// Quality of Life (QOL) Section:
//
//////////////////////////////////

//
// (1) Setup the Quality of Life array of column names:
//
$qol = array( "smoke_start"   , "smoke_end"   , "smoke_amount"   ,
              "alcohol_start", "alcohol_end", "alcohol_amount",
              "vitamin_start", "vitamin_end", "occupation","sun_exp" );
//
// (2) Retrieve values:
//
$qol_values = array();
retrieveIndividualValues($individualID,&$qol,&$qol_values);
//
// (3) Create form section:
//
createFormSection("Quality of Life",3,&$qol,&$qol_values);
```

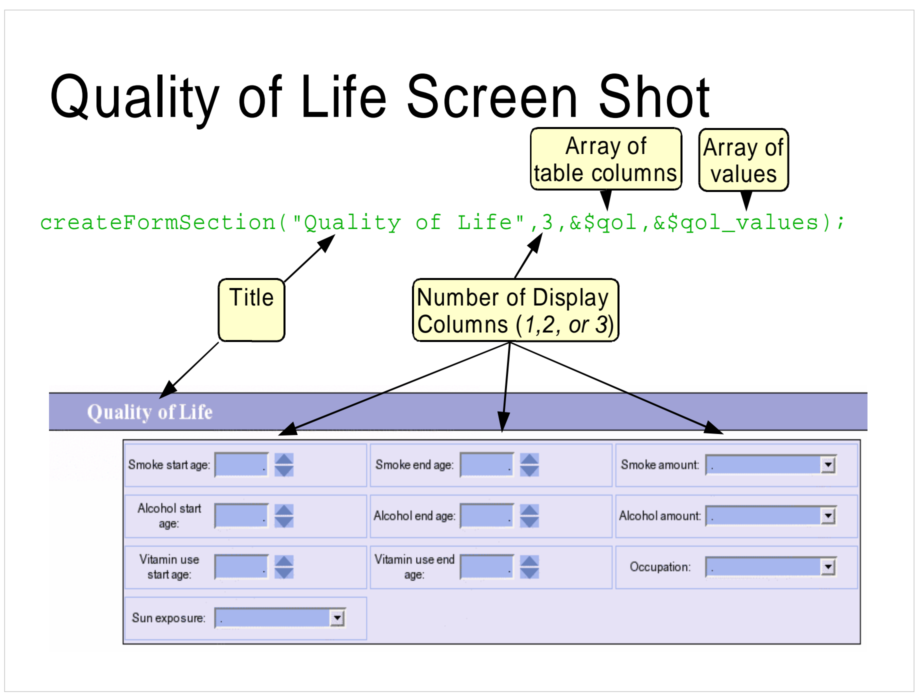
1 2 3

The layout manager makes it trivially easy for a Gladiator developer to create and populate a section of a form:

1. First, one creates an array containing the names of the columns one wishes to appear in that section of the form.

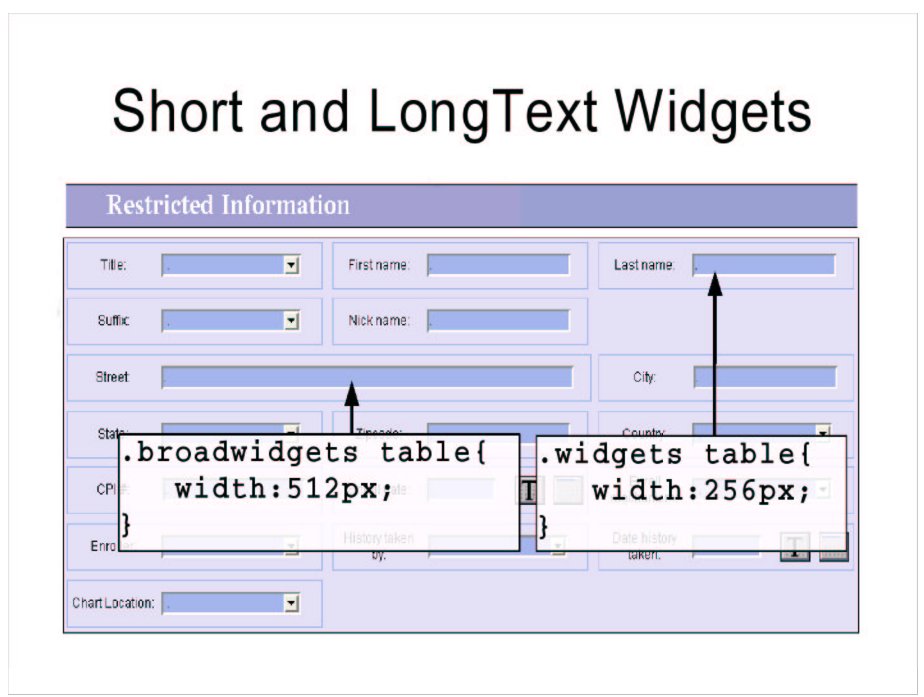
2. Secondly, if the form needs to display data for an existing record (say, of an individual), one retrieves the relevant values and stores them in an array.

3. Finally, one just calls **createFormSection()**.



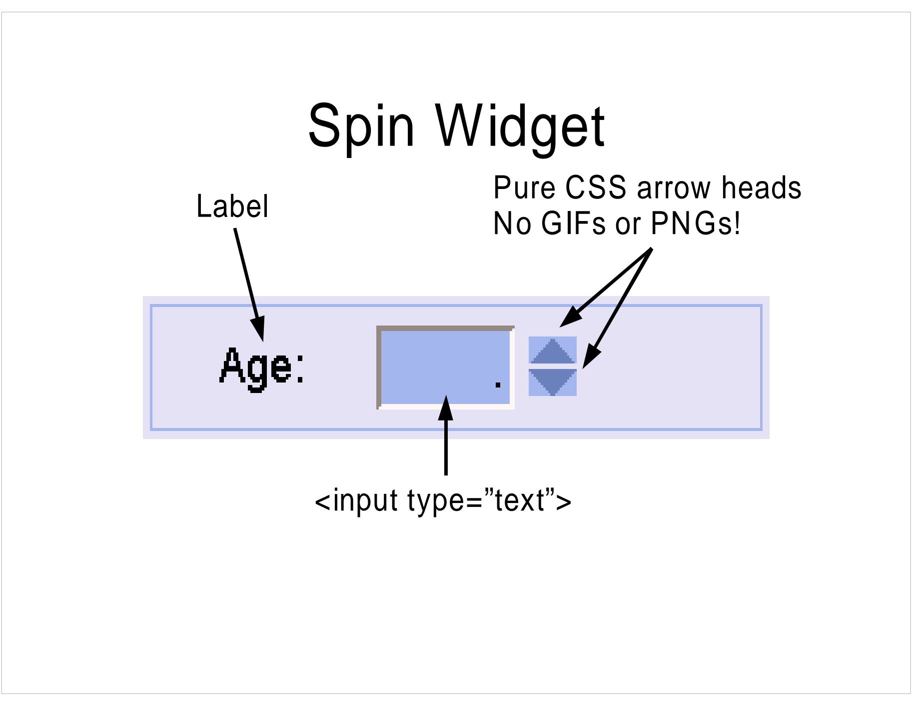
The createFormSection() function takes four parameters:

(1) A title for the section.
(2) The number of display columns.
(3) The array of column names.
(4) An optional array of column values.



The widgets share a core set of CSS styling attributes.

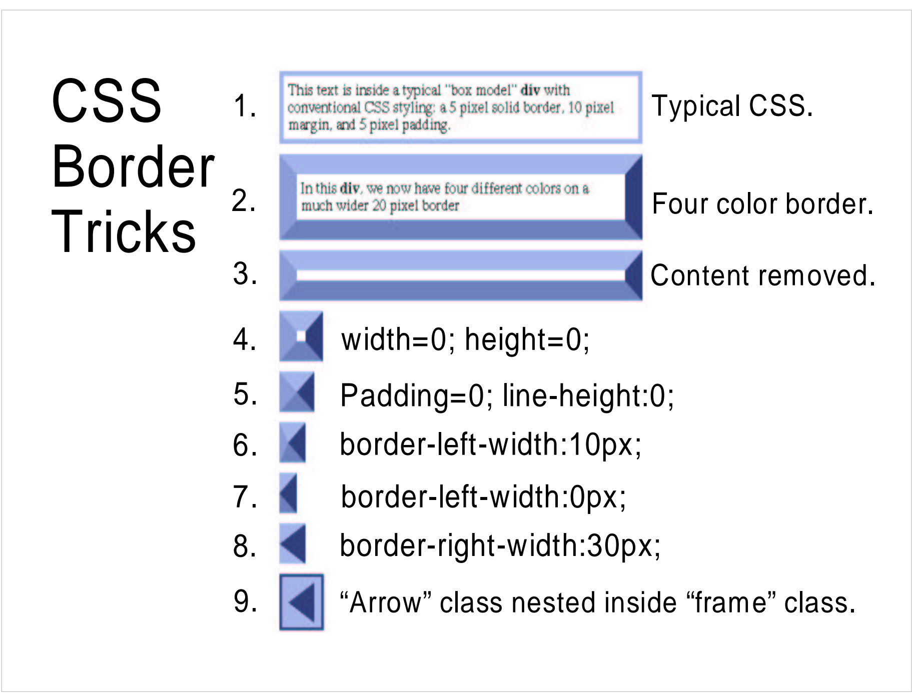
The *only* difference between the *short* and *long* (i.e., broad) text widget is in the class definitions of the CSS styling attributes.



Now let's take a look at a widget in detail. I want to walk through the construction of the *spin widget*, not because it is complex, but actually for the opposite reason: *because it is so simple!* This is a small, but very good illustration of writing code that is *good for both the user and the developer!*

The gladiator widgets use an HTML table as the basic layout container. The table and table elements are then styled using CSS.

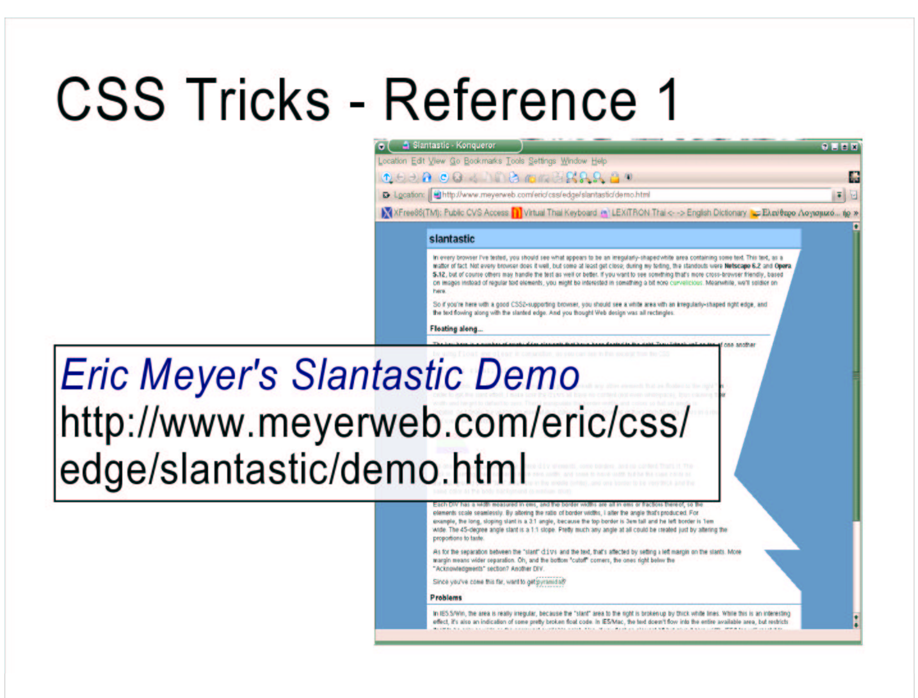
For the spin widget, the table contains three cells: one for the *label*, one for the *<input type="text">* element, and one for the *up and down arrows* of the spin control. Our first trick is to use pure CSS for the arrows!



Arrow heads are constructed using CSS border properties. The trick is to color the four borders of a box appropriately, reduce the content area of the box to zero, and make the borders thick on three sides and zero-width on the fourth side.

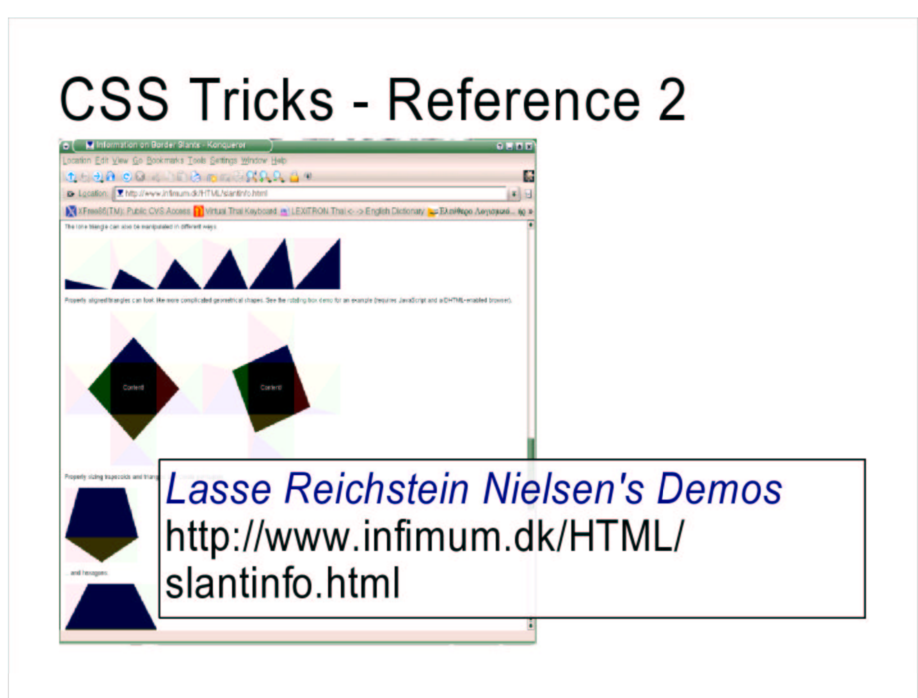
By using this trick, we avoid using bitmapped images and are able to display triangular arrow heads at any scale and color!

We essentially achieve a scalable arrow head without SVG. In the future when browsers provide native SVG rendering, using SVG arrow heads may prove a more elegant solution. But, for now, this approach achieves excellent results!

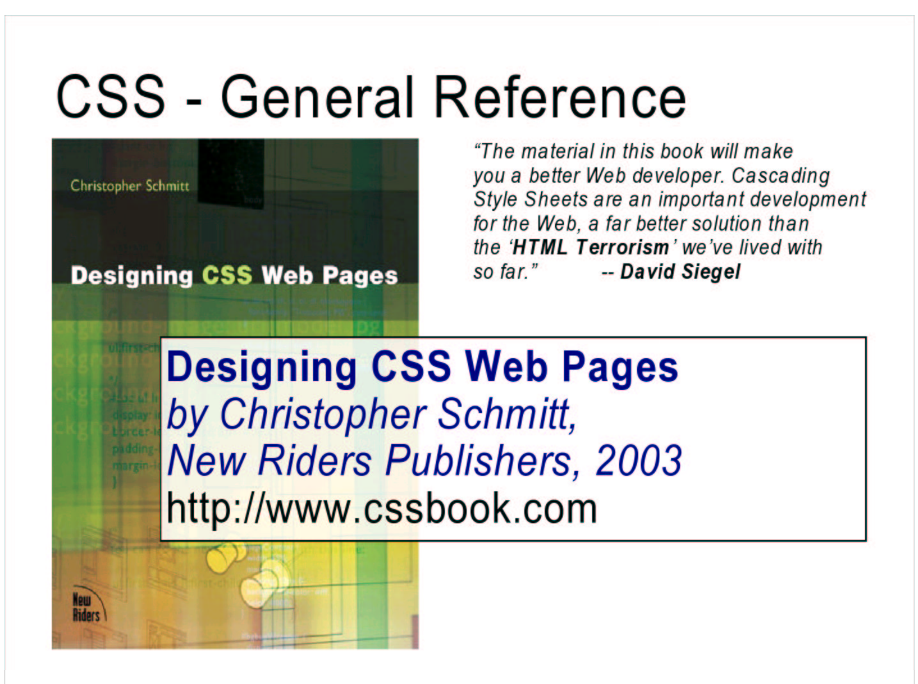


I don't know if we really “invented” the CSS arrowhead trick, but I do think we found a good purpose for it!

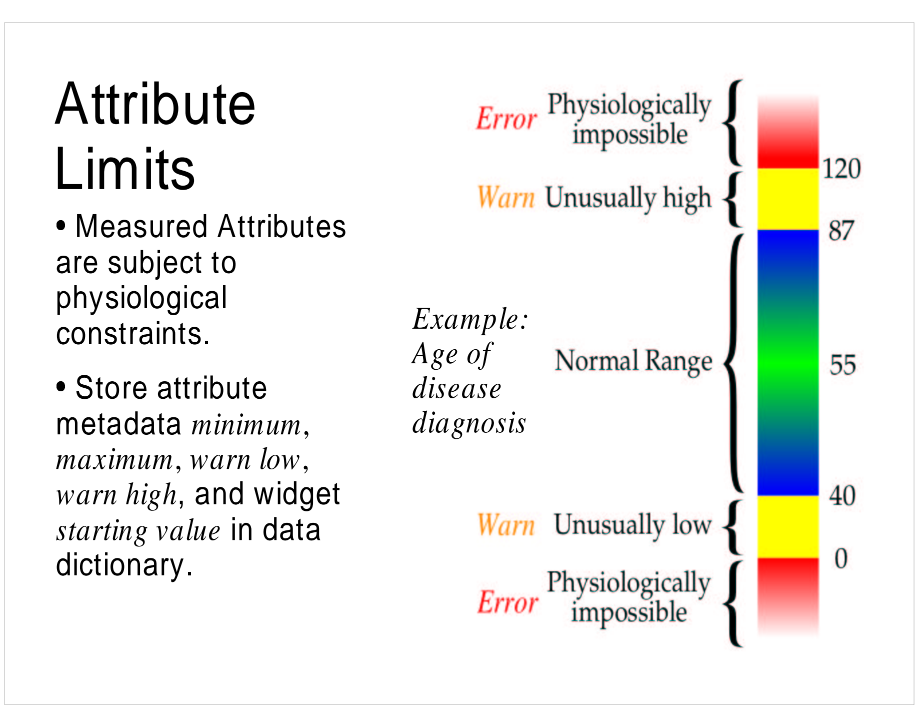
When we were researching CSS for this project, I got the idea partly from Eric Meyer's “Slantastic Demo” ...



... and partly from Lasse Reichstein Nielsen's CSS demonstration pages. Nielsen's pages are really “wild” and worth taking a look at! If you do, don't neglect the animated star and valentine's heart!



A very good reference for more general CSS design is Chistopher Schmitt's book, **Designing CSS Web Pages** (New Riders Publishers, 2003).



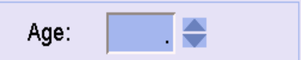
Let's get back to the spin widget:

All continuously-valued attributes measured or recorded for a patient are subject to **physiological constraints on the range of acceptable values.**

For each continuously-valued attribute in our data dictionary, we can store the appropriate meta data for use by the **Spin Widget**: *minimum, maximum, warn low, warn high*, and *starting value*.

This meta data is used by the widget's associated Javascript.

Spin Widget



Function createSpinWidget() Parameters

```
$label        : Label for the widget
$name         : Name & ID attr. of <input> tag
$min          : Minimum allowed value
$max          : Maximum allowed value
$warnlow      : Trigger warning below this value
$warnhigh     : Trigger warning above this value
$initial=""   : Initial value: def=(max-min)/2
$inc=1        : Increment value: def=1
$prec=0       : Number of decimal places: def=0
```

As mentioned, “widgets” are the combination of form elements and supporting HTML, CSS, and DOM-based Javascript validation code.

After experimenting extensively, we decided that our “widgets” would be layed out in tables with CSS used for styling all of the elements.

(Pure CSS class-based widgets without using layout tables may be possible, but we thought that would be too difficult).

We try to keep our code as simple as possible. To show you just how simple it is, Let's go through the PHP code for the **createSpinWidget()** function.

Spin Widget Code (1): Label

```
if($initial == "")
  $initial = floor(($max-$min)/2);

echo " <!-- START SPIN WIDGET $name -->\n";

echo "       <div class=\"widgets\">\n";
echo "        <table>\n";
echo "         <tr>\n";
echo "          <td class=\"tdwidth1\">\n";
echo "           $label:\n";
echo "          </td>\n";
```

In order to ensure that the HTML generated by PHP is readable, we do two things. First, we start and end each widget with a comment. Notice that the comment contains **$name**. **$name** is normally just the table column name.

Secondly, we indent each nested HTML element by one space (tabs are too big, so we use spaces).

Here you can see the label for the spin widget is placed into a **<TD>** element.

Spin Widget Code (2): Input

```
echo "           <td class=\"tdwidth2\">\n";
echo "            <input
                   class=\"spinwidget\"
                   type=\"text\"
                   name=\"$name\"
                   id=\"$name\"
                   value=\".\"
                   onchange=\"return
                    checkRange(
'$name',$min,$warnlow,$warnhigh,$max)\">\n";

echo "           </td>\n";
```

Here's the **INPUT** element for the widget. Note the default value, “.”.

Spin Widget Code (3): Up Arrow

```
  echo "          <td>\n";
  echo "           <div
class=\"csswidgetuparrow\"
onclick=\"incrementTextValue
('$name',$initial,$inc,$prec,$max);
return checkRange
('$name',$min,$warnlow,$warnhigh,$max)
\">\n";
  echo "            <!-- This is the up-
pointing arrow head -->\n";
  echo "           </div>\n";
```

Here's where we use CSS creatively!

The up- and down-pointing arrow heads are just empty **<DIV>** elements with stylized CSS border properties as described earlier.

Of course, we also provide a Javascript function to respond to the **onClick** event. The **incrementTextValue()** function won't allow a value greater than **$max** (the user can of course still type an excessive value directly).

The **checkRange()** function decides whether the background of the text input element should change to *yellow* (*warning*) or *red* (*out of range*).

Spin Widget Code (4): Down Arrow

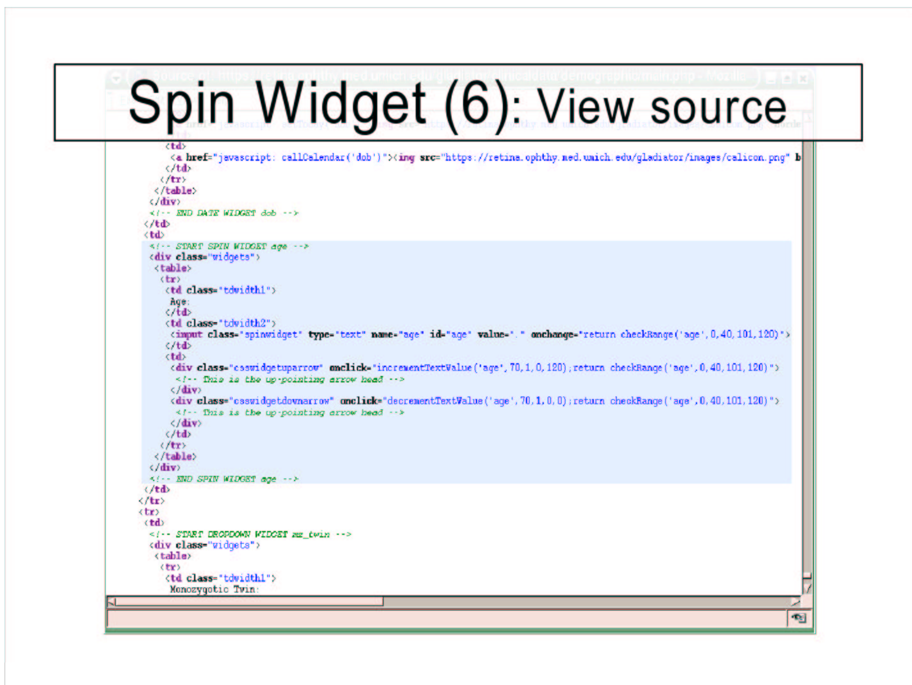
```
  echo "           <div
class=\"csswidgetdownarrow\"
onclick=\"decrementTextValue
('$name',$initial,$inc,$prec,$min);return
checkRange
('$name',$min,$warnlow,$warnhigh,$max)
\">\n";
  echo "            <!-- This is the down-
pointing arrow head -->\n";
  echo "           </div>\n";
```

The down-pointing arrow is naturally almost identical to the up-pointing arrow.

Spin Widget Code (5):Finish

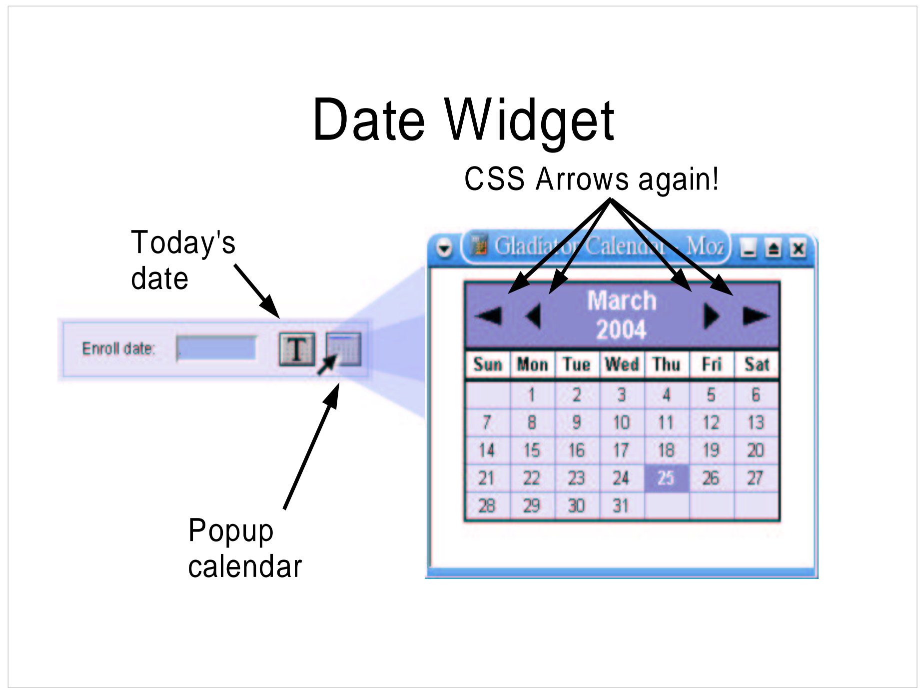
```
  echo "           </td>\n";
  echo "          </tr>\n";
  echo "         </table>\n";
  echo "        </div>\n";
  echo "        <!-- END SPIN WIDGET $name
-->\n";
}//createSpinWidget
```

End of the **createSpinWidget()** code.



The HTML generated by PHP has a starting and ending comment, and single-spaced indentation.

This makes the code easy for the developers to read and debug.



The date widget provides another example of how we looked closely at:

- What would be best for the user?

and also:

- What would be best for us as developers?

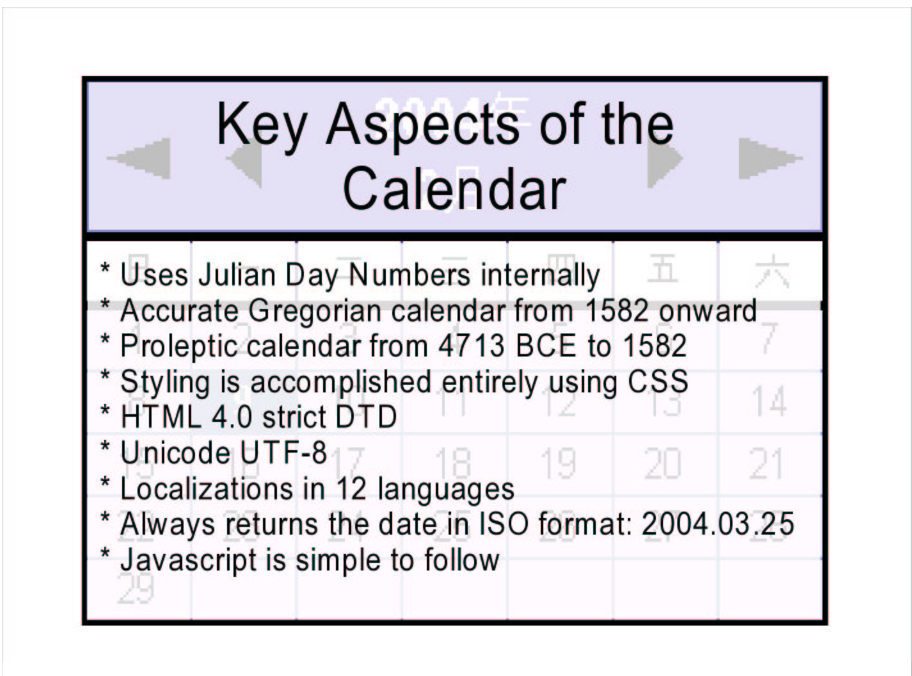
Note the use of CSS arrows once again.

Calendar: Us vs. Them

Ours	Theirs
* Standards based: works in Moz., Konq., Opera, IE ...	* `if(ie){...}` `else if(ns){...}`
* Lines of Code * HTML: 85 LOC * Javascript: 520 LOC (17.8 KB) * CSS: 167 LOC	* Javascript Code: Completely obfuscated in 2 LOC in a 25.8 KB file (45% larger than ours)
* Basic feature set, but works for all years.	* Overloaded with features, but doesn't even work properly for the year 1954.

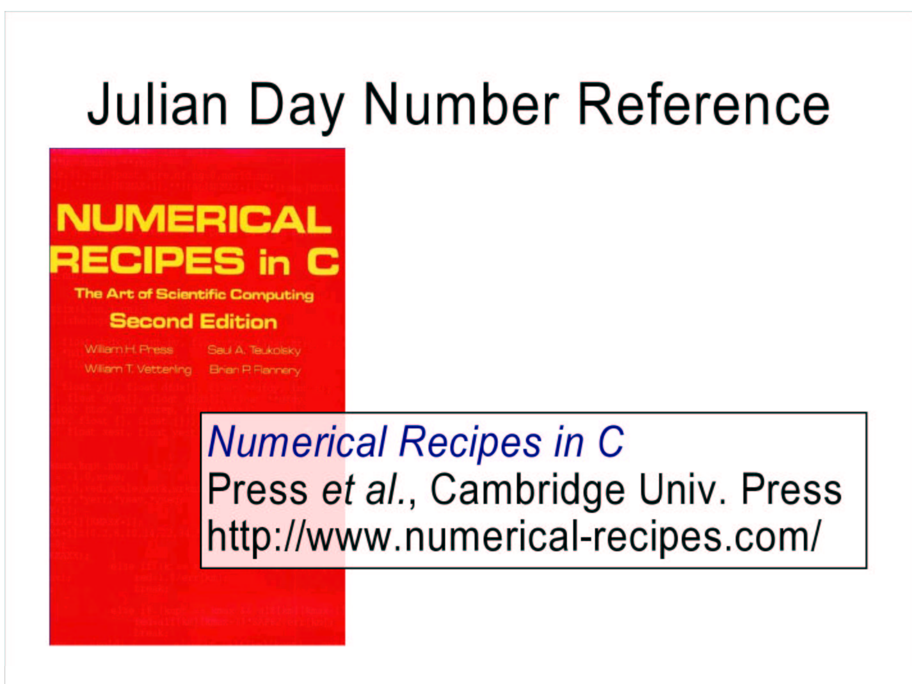
In a previous sample-tracking project called **Stella**, we had used an "off-the-shelf" Javascript pop-up calendar written by somebody else. The Javascript code for that calendar was completely riddled with **IE-** and **Netscape-**specific code, and was obfuscated into 2 very long lines of code! There was no way to make heads-or-tails of it. That code did not work properly for birthdates prior to 1970! I also had some issues with the license terms.

I knew that I could write something *simpler*, *better*, and *standards-based*.



Key aspects of the calendar are:

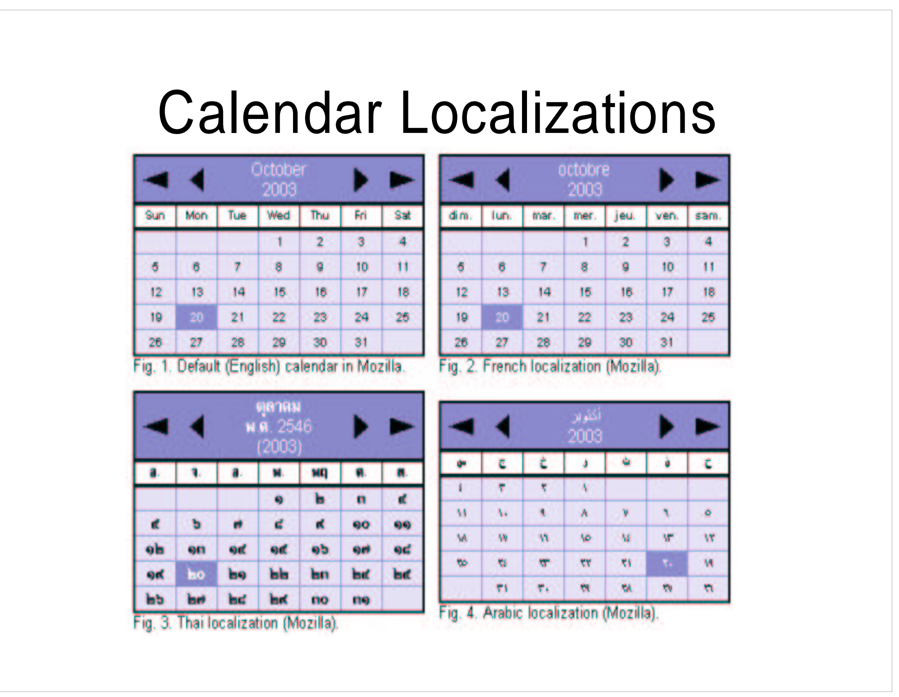
- Uses Julian day numbers internally.
- Accurate Gregorian calendar from 1582 on.
- Proleptic calendar from 4713 BCE to 1582.
- All styling is accomplished via CSS.
- Uses HTML 4.0.1 strict DTD.
- Localized in 12 languages.
- Returns date in ISO format: 2004.03.25.
- Javascript is simple to follow.



Algorithms for converting between Julian day numbers and gregorian or Julian dates are described in the **Numerical Recipes** books by Press, Flannery, Teukolsky, and Vetterling (Cambridge University Press).

The book is now available in (at least) versions for **C, C++, Fortran 77,** and **Fortran 90**. You can also browse through it online.

This is an indispensible reference for many kinds of scientific programming.



Use of Unicode UTF-8 transformation format made it easy to localize the calendar with minimal changes to the code.

* HTML is created using ...

Calendar Javascript

```
calendarWindow.document.write( string );
```

* The calendar calls ...

```
window.opener.generateCalendar
( year,month,day,destinationId );
```

...rather than calling `generateCalendar()` directly.

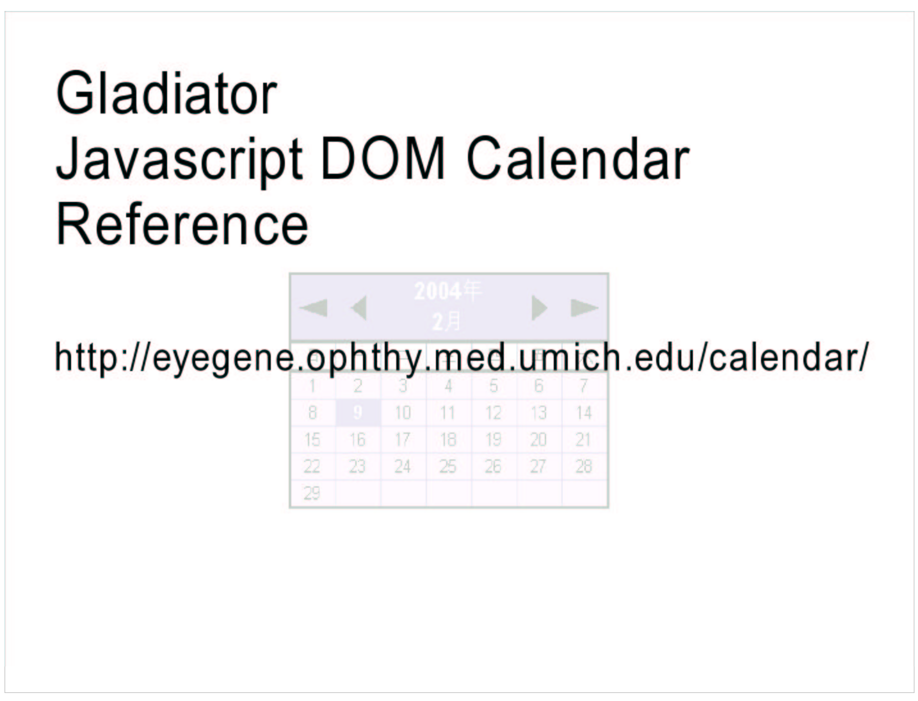
*This is done in order to preserve ...

```
window.opener.document.getElementById
( destinationId );
```

... in the `setDate()` call.

The HTML for the popup calendar is created using the **document.write()** method.

A very subtle requirement is that the calendar must call **window.opener.generateCalendar()** rather than **calendarWindow.generateCalendar()** in order to preserve the ability to call **window.opener.document.getElementById (destinationID)** in then **setDate()** call.



You can download our Javascipt DOM Calendar from our website:

- **eyegene.ophthy.med.umich.edu/calendar/**

Client vs. Server: Zipcode Validator

Server
Zipcodes of all 50 states are stored in database

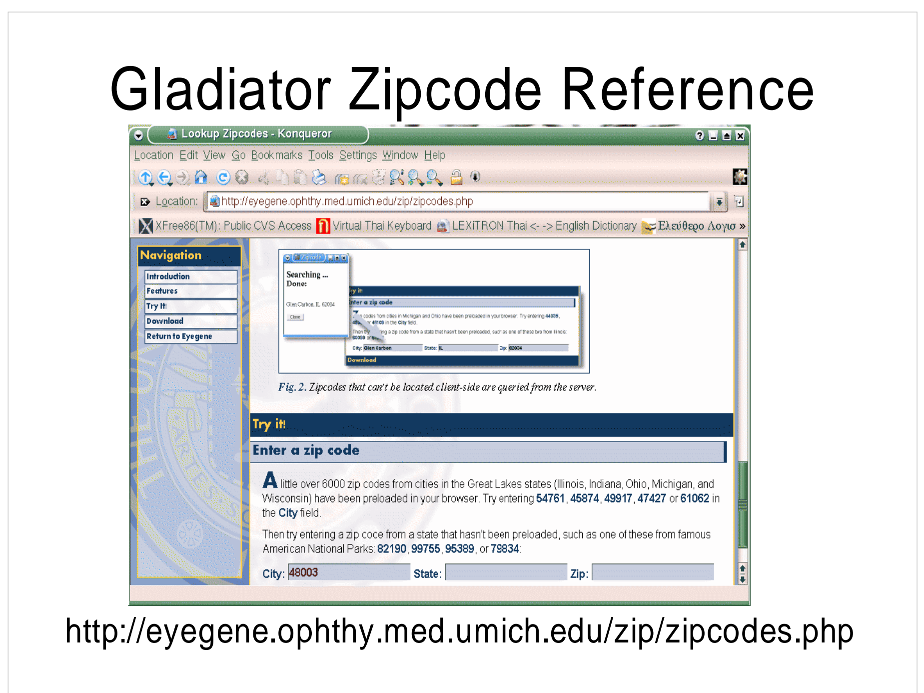
Client
Only most frequently used zipcodes are initially sent to the client

When designing a web-based system, one often has to decide which data should be processed on the server side, and which data should be loaded and processed on the client side.

Server-side processing is ideal when you have a lot of data, but network latency can sometimes be an issue.

Client-side processing using Javascript avoids network latency issues, but is only useful if one doesn't need to load too much data onto the client.

Our solution for entering address information (city, state, zipcode) in Gladiator employs a “hybrid” model where some zipcodes --those most frequently encountered (in-state and neighboring state zipcodes)-- are loaded when the page loads. Searches for out-of-state zipcodes require a server query.

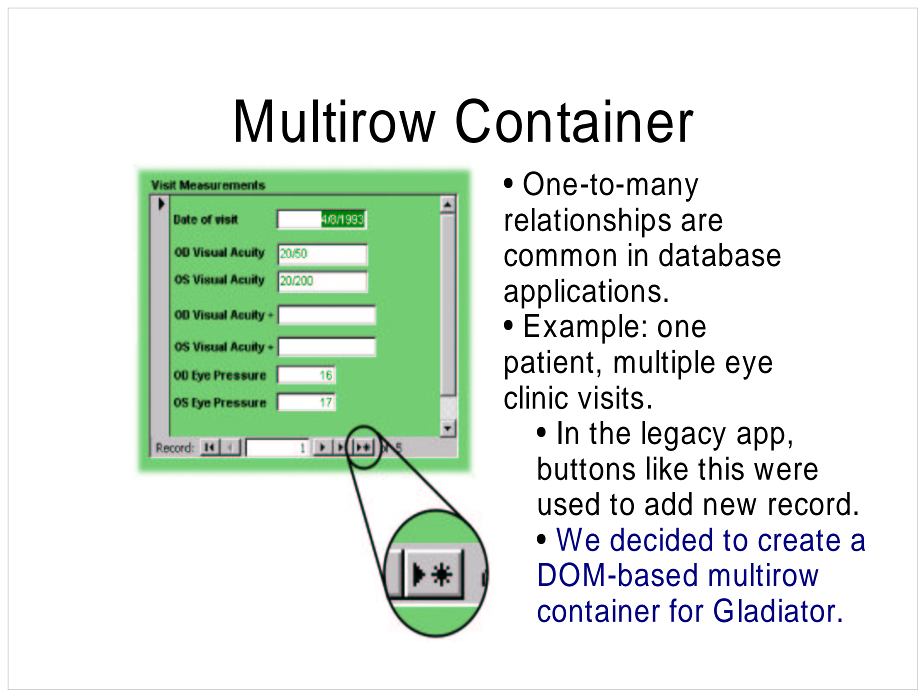


Here you see our zipcode demonstration page.

When you enter a local zipcode (from Michigan or a neighboring state in the Great Lakes Region), the corresponding city and state are found “instantly” via our Javascript array lookup.

When you enter a non-local zipcode from out-of-state, a popup form is created and submitted by Javascript. The returned results are automagically plugged back into the form.

DOM

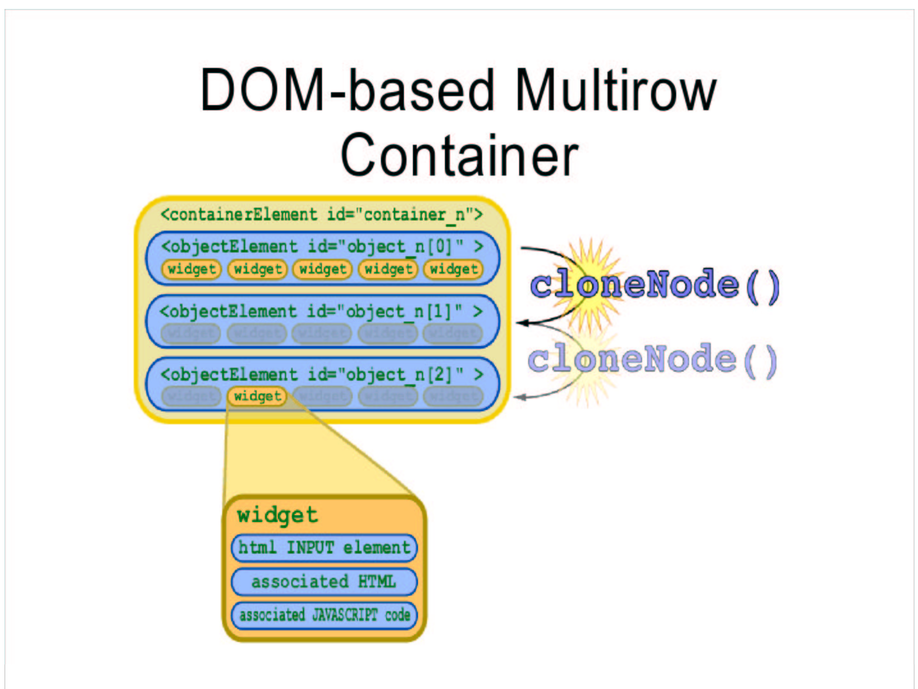


One-to-many relationships are common in database applications.

In the Gladiator project, we wanted to be able to enter a number of eye conditions on one patient all at once and only have to press the **Save** button once.

The original legacy application, written as a thick client in Microsoft Access had this ability.

To do this, we realized that we would need a web form in which we could, after completing one row, add a second, third, or fourth row as we continued to enter additional data.

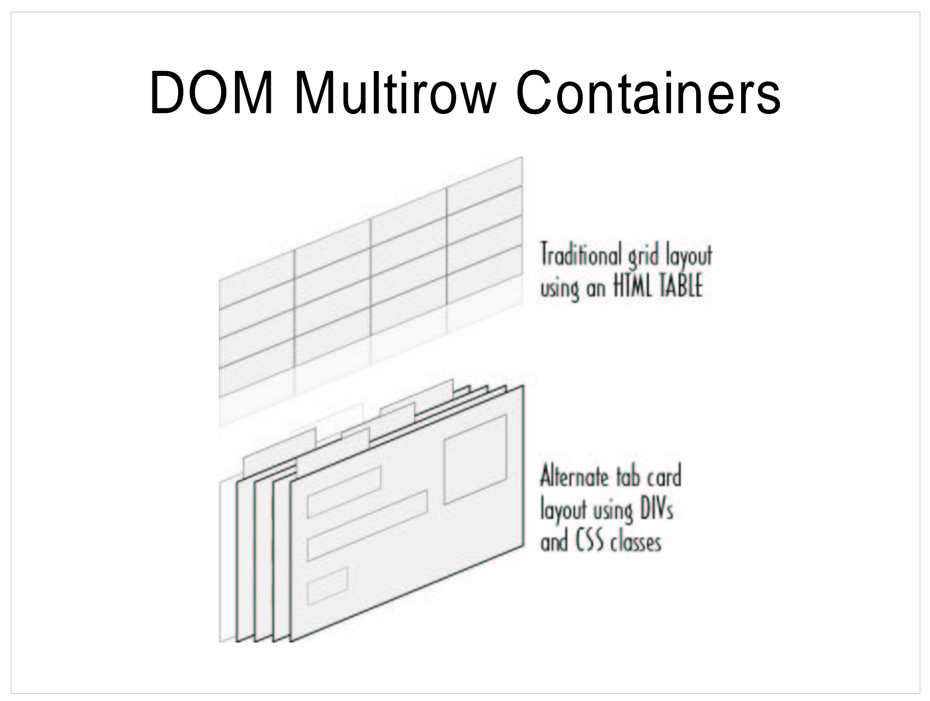


Conceptually, we have a **container** which holds a number of **objects**.

The **objects** can be **<TR>** rows or some more "exotic" **<DIV>** element that you style using **CSS**.

The **objects** are themselves containers for the **widgets**.

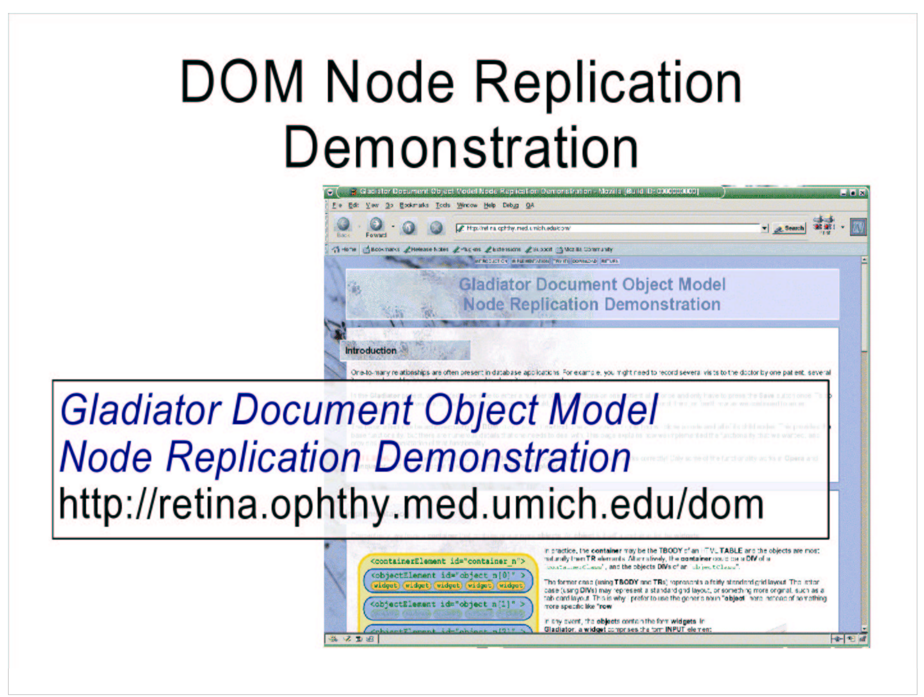
The basic effect can be achieved using the **DOM**-based Javascript **cloneNode()** method, but there are numerous other details that we also needed to deal with.



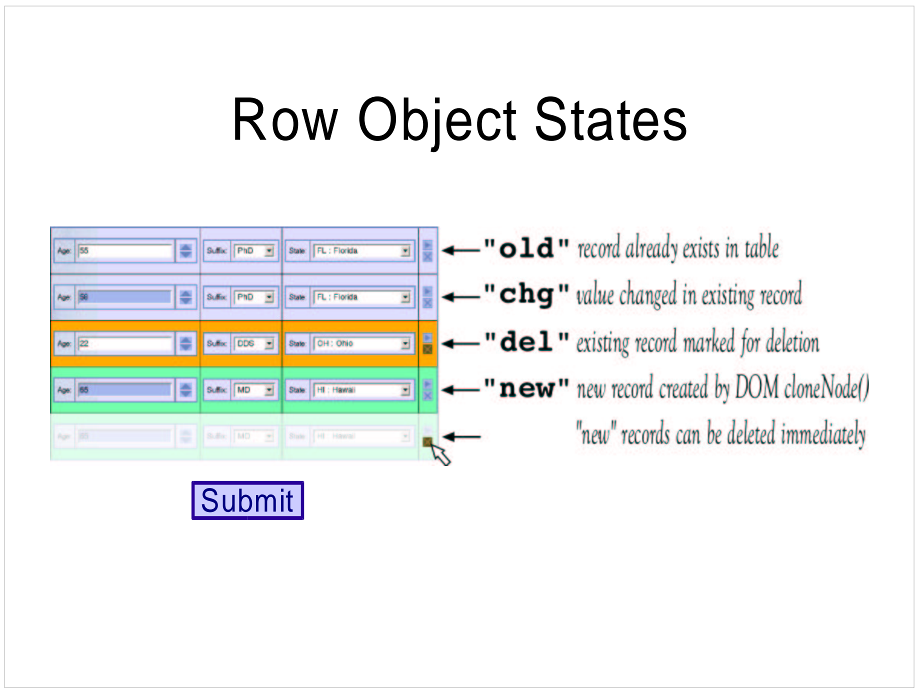
In practice, the container may be the **TBODY** of an **HTML TABLE** and the objects are most naturally then **TR** elements. Alternatively, the container could be a **DIV** of a "**containerClass**", and the objects **DIV**s of an "**objectClass**".

The former case (using **TBODY** and **TR**s) represents a fairly standard grid layout. The latter case (using **DIV**s) may represent a standard grid layout, or something more original, such as a tab card layout.

This is why I prefer to use the generic noun "**object**" here instead of something more specific like "**row**".



Our Gladiator Document Object Model Node Replication Demonstration page provides a detailed discussion of what's required to use DOM replication techniques in a real application.

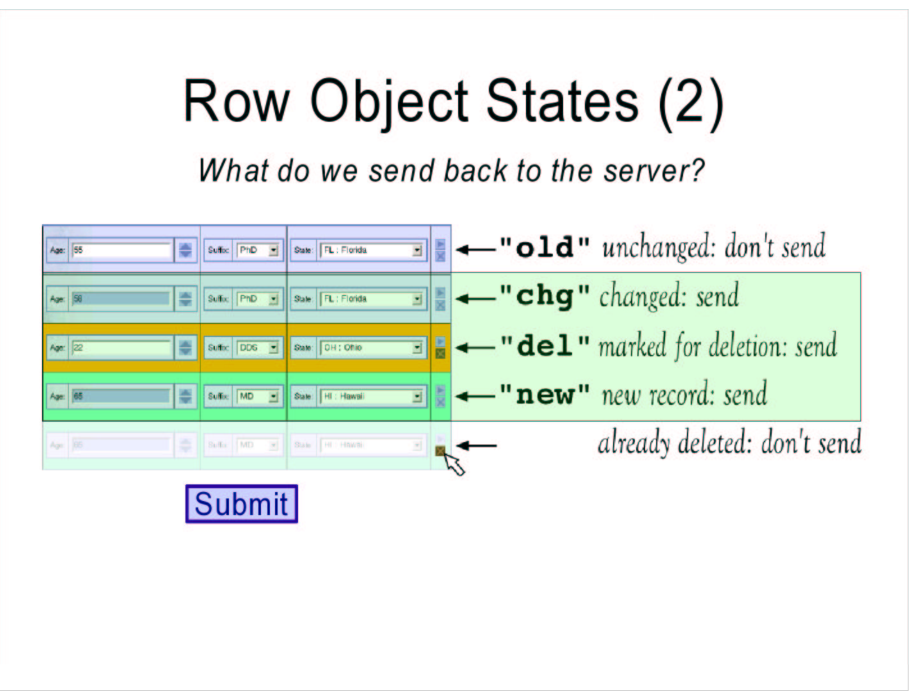


To show how this works, we'll assume a standard table-based grid layout with rows as records. If there are **existing records** in the table, we have **PHP** create rows with a status attribute set to "**old**". If there are **no existing records**, then **PHP** creates a single row with a status attribute of "**new**".

In either case, the first row is used as the template node which will be cloned by a call to cloneNode(). Clicking on the ">" arrow head creates a new row marked with a status attribute of "**new**".

If the user clicks on an "**X**", the Javascript code determines whether the row is "**new**" or "**old**". New rows get deleted immediately, but "**old**" rows get marked (visually in orange) since the actual deletion will need to be processed on the server side.

But how are changed rows detected? Values for the "**old**" rows are actually loaded into the grid by a Javascript **onLoad()** function on the **BODY** tag. When the user presses **SUBMIT**, the Javascript compares the values in the Javascript array against the values in the grid elements. This solution avoids having to have **onChange()** calls on every widget.



"Old" rows that remain unchanged don't need to be sent back to the server: we just delete the DOM nodes of all "old" rows.

As a result, only the data of the remaining "chg" (changed), "del" (marked for deletion), and "new" rows get sent back.

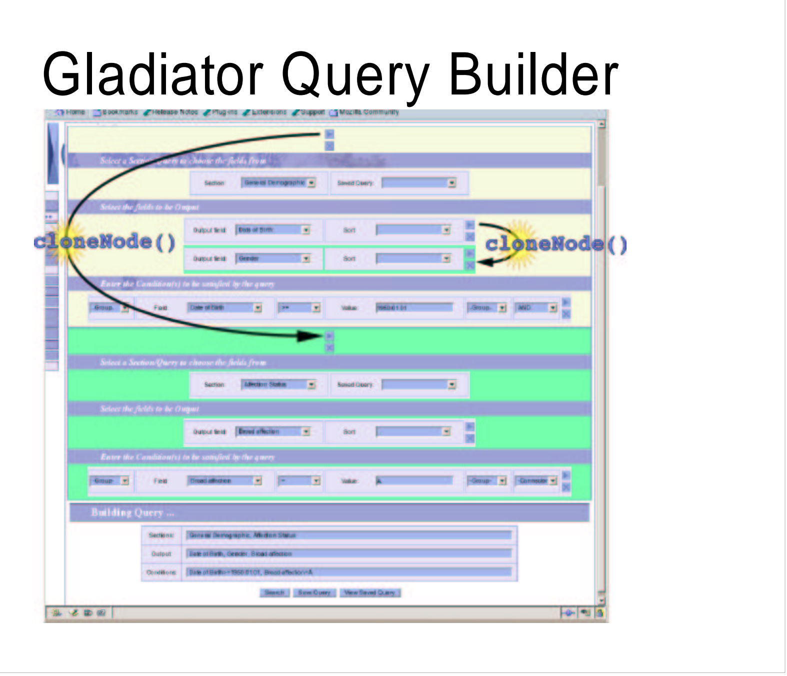
Recall that each widget is assigned a NAME and ID containing an array subscript in square brackets. As a result, the data sent back to the server in the POST is interpreted by PHP as a series of sparse (or partial) arrays.

Row Object States (3)

Server-side processing

```
foreach($_POST["status_1"] as $key -> $value)
{
   switch($value){
   case "del":
      // Process entry marked for deletion ...
      deleteEntry(...);
      break;
   case "new":
      // Add new record to table ...
      addEntry(...);
      break;
   case "chg":
      // Update table with new values ...
      updateTableValues(...);
      break;
   }
}
```

On the server side, we hence use PHP's **foreach** statement to loop through the partial arrays. (We could not use an ordinary **for** loop, because some of the index values may be missing).



Recall that we originally wrote the DOM node replication Javascript code in order to use it on the clinic visits screen in Gladiator.

Not long after Ritu had completed the clinic visits screen, she had the brilliant idea to use our DOM node replication code to create an easy-to-use query/report builder, shown here.

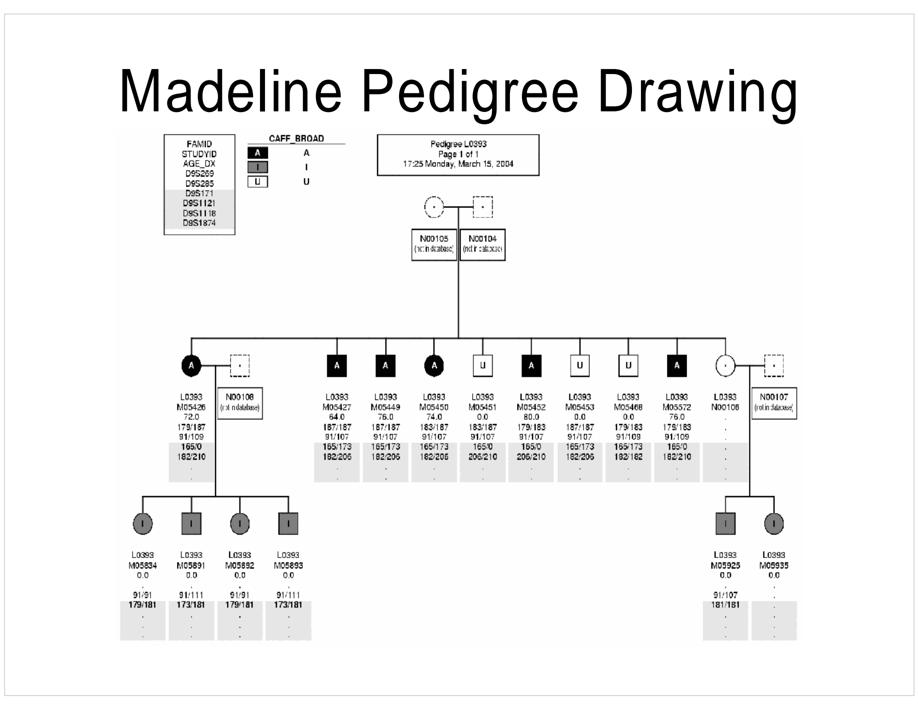
In this form, one first chooses the “section” -- this is equivalent to selecting a table to query from. Within this section/table, one can then choose the columns.

Multiple columns can be chosen from one table, and multiple tables can be selected to construct a query. The DOM node replication code is indispensible to making this all work.

This is a good example of how, once you have written some very generic code, you end up finding many new uses for it that never occurred to you originally.

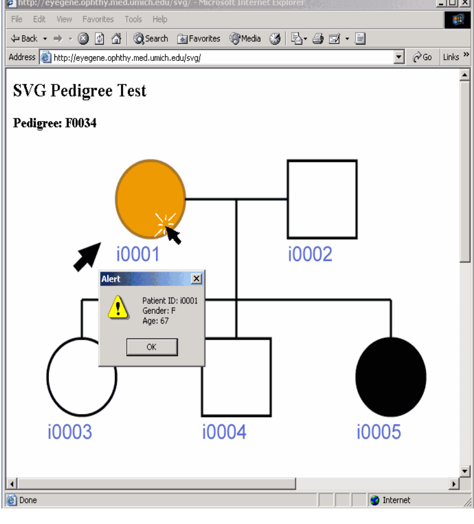
SVG

"Scalable Vector Graphics" is the World Wide Web Consortium's new open standard for vector graphics. SVG promises to provide new ways for developers to use visual information in the creation of interactive experiences for users.



This is a Postscript-based drawing of a human pedigree generated by Madeline. In addition to showing the relationships between people in the family, some clinical and genetic data are shown (age of diagnosis and alleles typed for specific markers on Chromosome 9).

In Madeline, pedigree drawings are generated directly from data in the database. While Madeline's pedigree drawings have proved to be a very useful feature of the program, the Postscript format is not an "interactive" format, and the drawings are best viewed printed on paper.



SVG promises to provide a single format that is appropriate to both on-screen interactive graphics, and off-screen print media.

Here's a quick pedigree demo I assembled. Clicking on any individual in the SVG-based pedigree drawing pops up detailed information about that individual.

We plan to incorporate SVG-based pedigree drawing into an online pedigree editing module in Gladiator. This demo is just a simple proof of concept. In the actual application, the user will have buttons labeled “brother”, “sister”, etc., available for adding new people to the pedigree. Clicking on individuals will also provide access to forms available in Gladiator.

SVG Code Sample

```
<ellipse
   onclick="showinfo(evt,'i0001','F',67)"
   onmouseover="changeColor(evt,'#e90')"
   onmouseout="changeColor(evt,'#000')"
   class="affected"
   cx="200"
   cy="70"
   rx="50"
   ry="50"
   id="i0001" />
<rect
   onclick="showinfo(evt,'i0002','M',68)"
   onmouseover="changeColor(evt,'#aaf')"
   onmouseout="changeColor(evt,'#fff')"
   class="normal"
   width="100"
   height="100"
   x="400"
   y="20"
   id="i0002" />
```

SVG has many appealing aspects. Because it is based on XML, SVG code is very similar to (X)HTML code.

The code snippet here shows the definition of the first two individuals --the parents-- from the SVG pedigree demo shown in the previous slide.

You can see here (in orange) the calls to a Javascript function attached to the **onClick** event.

You can also see (in blue) the use of **CSS** classes called “normal” and “affected” for styling of the pedigree icons.

Note that we could have used **Flash** instead of **SVG**, but by utilizing **SVG**, we extend our knowledge investment in open technologies.

Now we are going to discuss the security model and user roles in Gladiator.

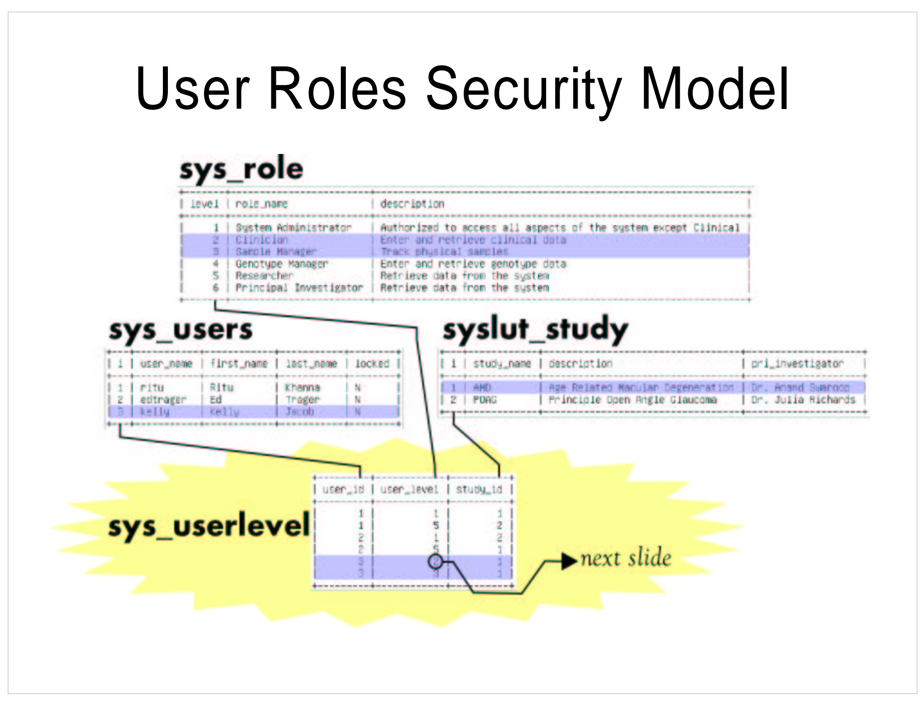
Our approach to security is multi-tiered:

1. At the OS level, the server firewall will be set up to only permit HTTPS requests from clients within known subnets in the Kellogg Eye Center. This approach was used in Cicada, Gladiator's predecessor, as well.
2. Access to Gladiator is via secure HTTPS.
3. Users are issued usernames and passwords which are encrypted in the database.

Studies, Users, and Roles

- Research labs conduct multiple studies.
- Clinicians and researchers perform specific roles within each study.
- One person might perform one set of roles in one study, and a different set of roles in a different study.

Because research labs conduct multiple studies, and clinicians and researchers may perform different roles in different studies, it was important to design Gladiator to intelligently handle access requirements so that personnel do have access where they need it, but don't get access to portions of the system or studies that they don't have authority to view.



User Roles Security Model

sys_role

level	role_name	description
1	System Administrator	Authorized to access all aspects of the system except Clinical
2	Clinician	Enter and retrieve clinical data
3	Sample Manager	Track physical samples
4	Genotype Manager	Enter and retrieve genotype data
5	Researcher	Retrieve data from the system
6	Principal Investigator	Retrieve data from the system

sys_users

i	user_name	first_name	last_name	locked
1	ritu	Ritu	Khanna	N
2	edtrager	Ed	Trager	N
3	kelly	Kelly	Jacob	N

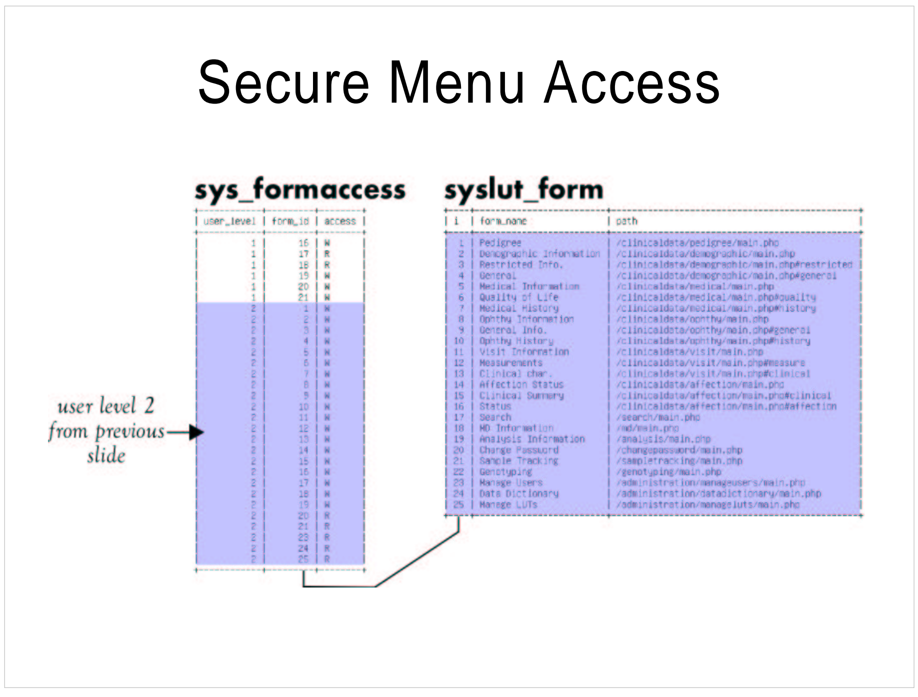
syslut_study

i	study_name	description	pri_investigator
1	AMD	Age Related Macular Degeneration	Dr. Anand Swaroop
2	POAG	Principle Open Angle Glaucoma	Dr. Julia Richards

sys_userlevel

user_id	user_level	study_id
1	1	1
1	5	2
2	1	2
2	5	1
3	2	1
3	3	1

The sys_userlevel table tracks which users perform what roles in what studies.



sys_formaccess

user_level	form_id	access
1	16	W
1	17	R
1	18	R
1	19	W
1	20	W
1	21	W
2	1	W
2	2	W
2	3	W
2	4	W
2	5	W
2	6	W
2	7	W
2	8	W
2	9	W
2	10	W
2	11	W
2	12	W
2	13	W
2	14	W
2	15	W
2	16	W
2	17	W
2	18	W
2	19	W
2	20	R
2	21	R
2	23	R
2	24	R
2	25	R

syslut_form

i	form_name	path
1	Pedigree	/clinicaldata/pedigree/main.php
2	Demographic Information	/clinicaldata/demographic/main.php
3	Restricted Info.	/clinicaldata/demographic/main.php#restricted
4	General	/clinicaldata/demographic/main.php#general
5	Medical Information	/clinicaldata/medical/main.php
6	Quality of Life	/clinicaldata/medical/main.php#quality
7	Medical History	/clinicaldata/medical/main.php#history
8	Ophthy Information	/clinicaldata/ophthy/main.php
9	General Info.	/clinicaldata/ophthy/main.php#general
10	Ophthy History	/clinicaldata/ophthy/main.php#history
11	Visit Information	/clinicaldata/visit/main.php
12	Measurements	/clinicaldata/visit/main.php#measure
13	Clinical char.	/clinicaldata/visit/main.php#clinical
14	Affection Status	/clinicaldata/affection/main.php
15	Clinical Summary	/clinicaldata/affection/main.php#clinical
16	Status	/clinicaldata/affection/main.php#affection
17	Search	/search/main.php
18	MD Information	/md/main.php
19	Analysis Information	/analysis/main.php
20	Change Password	/changepassword/main.php
21	Sample Tracking	/sampletracking/main.php
22	Genotyping	/genotyping/main.php
23	Manage Users	/administration/manageusers/main.php
24	Data Dictionary	/administration/datadictionary/main.php
25	Manage LUTs	/administration/manageluts/main.php

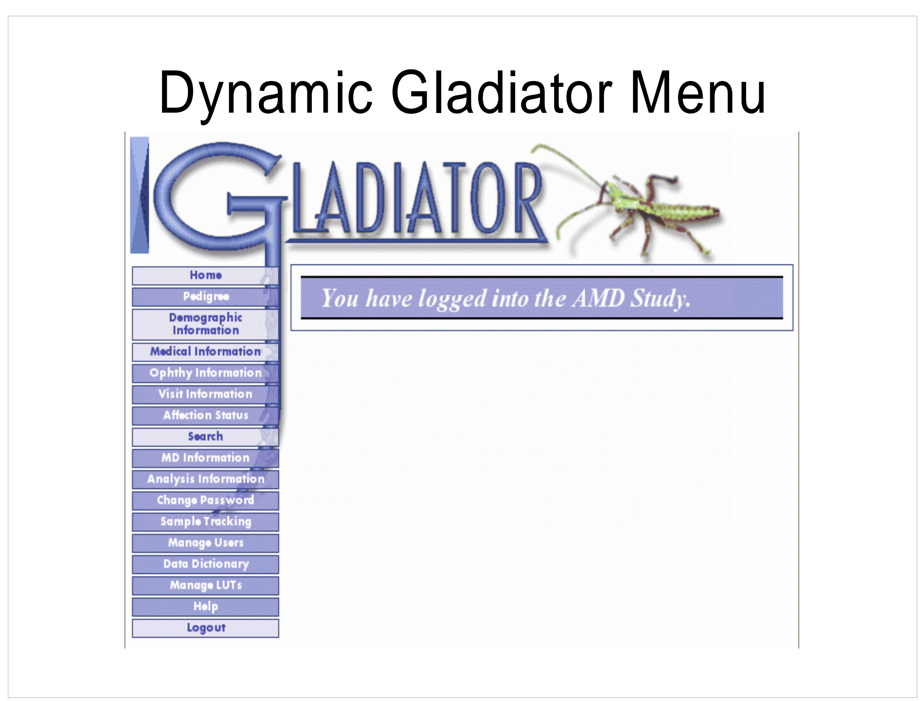
The sys_formaccess table tracks what forms are available to a user with a given user_level. For viewable forms, the table tracks whether the user has read-only or read-write permissions on data in that form. Forms not present for a given user_level are not available to that user.

Together with sys_formaccess, the syslut_form table is used to create a dynamic menu for the user.

Here are screen shots to show how this works.

After logging in, Kelly is presented with a table showing the roles that she has in various studies. In this case, Kelly is authorized to perform two roles on the AMD study.

Clicking with the mouse on the first row will allow her to perform the clinician role.



Here's the dynamic menu generated for the clinician role.

Also note the oblong rectangle to the left of the "G": clicking on this hides the menu, providing more screen real estate when needed.

UNICODE

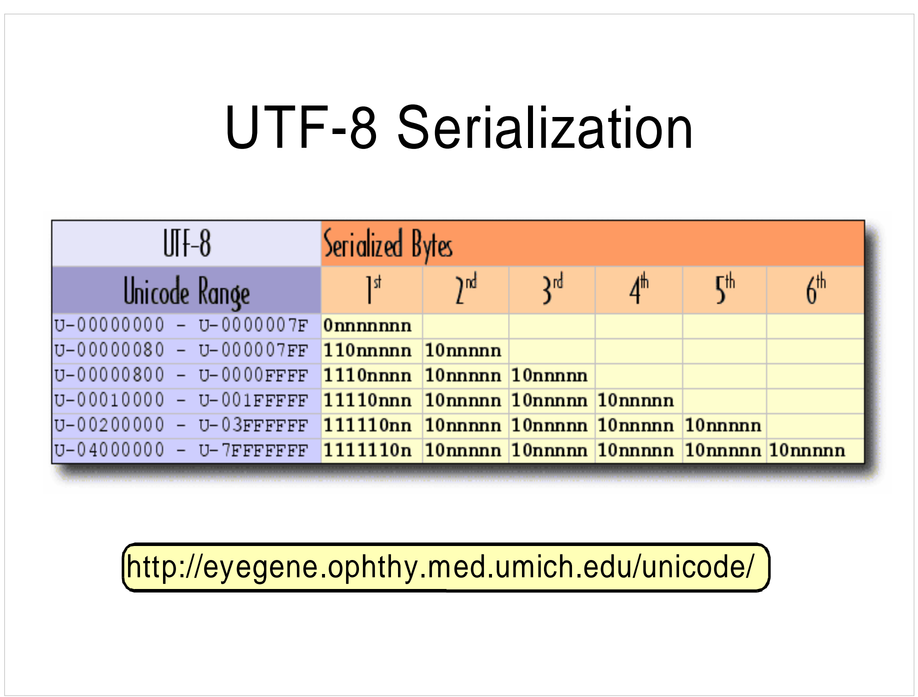
All Gladiator pages use the Unicode UTF-8 character set encoding.

The Unicode Consortium has allocated code blocks and code points for all of the worlds modern scripts, as well as quite a few historical scripts.

Unicode: UTF-8

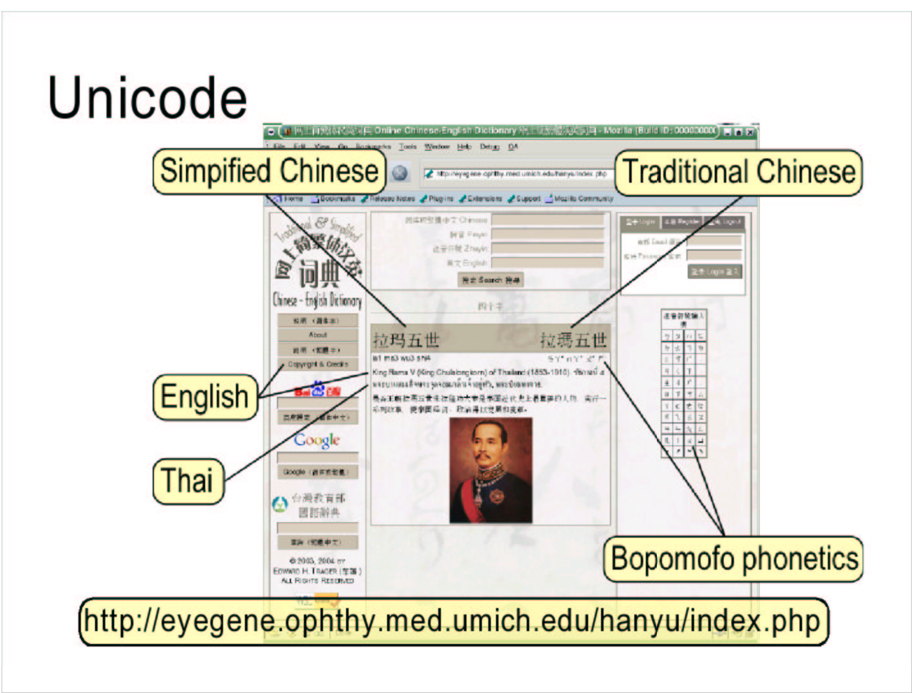
```
<!-- START HTML_BEGIN -->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.0.1//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
<html>
 <head>
  <meta http-equiv="Content-Type"
content="text/html; charset=utf-8">
```

UTF-8 is a serialization method for Unicode that is the de facto standard for encoding Unicode on UNIX-based operating systems, notably Linux. UTF-8 is also the preferred encoding for multi-lingual web pages.



UTF-8	Serialized Bytes					
Unicode Range	1st	2nd	3rd	4th	5th	6th
U-00000000 - U-0000007F	0nnnnnnn					
U-00000080 - U-000007FF	110nnnnn	10nnnnnn				
U-00000800 - U-0000FFFF	1110nnnn	10nnnnnn	10nnnnnn			
U-00010000 - U-001FFFFF	11110nnn	10nnnnnn	10nnnnnn	10nnnnnn		
U-00200000 - U-03FFFFFF	111110nn	10nnnnnn	10nnnnnn	10nnnnnn	10nnnnnn	
U-04000000 - U-7FFFFFFF	1111110n	10nnnnnn	10nnnnnn	10nnnnnn	10nnnnnn	10nnnnnn

In this method, ASCII code points occupy one byte. That is, the ASCII subset of Unicode serialized in UTF-8 is identical to ASCII. Unicode code points in the Basic Multilingual Plane above the ASCII range are serialized to two or three bytes (additional planes exist in Unicode, which can produce serializations of up to six bytes).



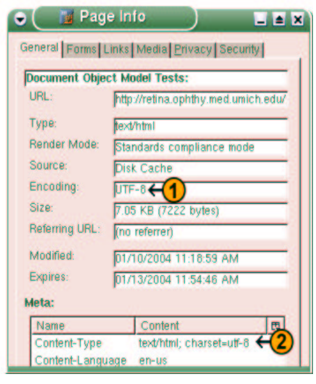
By using Unicode UTF-8 in your projects, you can easily support multiple languages. If you plan on releasing some or all of your code to the global internet audience, using UTF-8 is highly recommended.

Remember that the localization process is not simply an issue of translating strings into another language. Often there are subtle dependencies in the code that only become apparent when localization is done.

For example, in the **Chinese dictionary** project shown here, I encountered multiple issues with regular expression matching in both PHP's PCRE (Perl Compatible Regular Expressions) and MySQL's REGEX implementations.

Configuring UTF-8 in Apache 2

```
787 #AddDefaultCharset ISO-8859-1
788 AddDefaultCharset UTF-8
```



The standard specifies that web pages are served in **ISO-8859-1** (**Latin 1**) by default. In Apache's **httpd.conf** (around line 787), you can change this to **UTF-8**.

After restarting Apache, in Mozilla, select **View -> Page Info** to verify that pages are actually being transferred in UTF-8 encoding **(1)**.

The **META "Content-Type"** attribute **(2)** by itself is not enough for Mozilla (even though it is sufficient for other browsers like Konqueror).

Credits

- Ritu Khanna, *database programmer*
- Dr. Anand Swaroop, PhD.
- Dr. Julia Richards, PhD.
- Kari E. H. Branham, *genetic counselor*
- Beverly M. Yashar

- This project is supported by a generous grant from the **Elmer and Sylvia Sramek Charitable Foundation**, Chicago