Recall from last time:

SOA: Service-Oriented Architecture: the practice of composing programs from services.

SOAP: Simple Object Access Protocol: how services are built.

WSDL: Web Services Definition Language: describes how to build service clients.

WSAPI: Web Services Application Programming Interface: describes how to advertise services for search and discovery.
Advanced service composition

Limits of syntax and semantics.
The adapter pattern.
Transduction.
A dream that never happened
Advertise services (via WSAPI) in a "service marketplace"
Clients "subscribe" to services (for a fee).
Another word for subscription: "binding".
A client can change its mind about bindings: "on Tue, use google maps; on Thu, use MapQuest."

Logic behind the dream
Many services are very simple software front-ends to complex and difficult-to-maintain data repositories.
Data maintenance costs money.
Therefore, data access should cost money.
The service source code is irrelevant; it does nothing without the corpora.

Examples:
Location-aware services: map IP to Lat, Long.
   Theoretically: set by ISP.
   In practice: set by location-aware service provider.
Mapping services: translate Lat, Long to features at that Lat, Long
Theoretically: user-provided.
In practice: require substantive editing.
Reverse phone lookup: address from phone number
Already a cost-based service.
Problems with the dream

The problem of **ontology**: different service providers use different parameter names to mean the same thing.
The problem of **semantics**: different service providers can require different parameters.
There is no standard for invoking a specific kind of service, and a competitive advantage to varying the interface.
Consider the following two services:

A:
   Input: Latitude, Longitude
   Output: IPAddress

B:
   Input: Lat, Long
   Output: IP

Are these the same service?
How does one translate between them?

This is a problem of ontology and synonyms: when do two variable names mean the same thing?
A problem of semantics

Two services:

A:

Input: Lat, Long (in a spherical datum, e.g., a satellite grid)
Output: feature set, e.g., businesses located here

B:

Input: Lat, Long (in a spheroidal datum, e.g., GPS)
Output: feature set, e.g., businesses located here

The parameter names are the same: no problem of ontology and/or synonyms. The semantics of the parameters are different. (The same pair of numbers means a different location on the earth!)
Transduction

Refers to the process of transforming one object into another via a dataflow-like process.
Specific subcase: XML to XML (SOAP to SOAP) transformations.
Solution: XSLT (eXtensible Stylesheet Language Transformations)
A language specifically for transforming one XML document into another. Programs are "stylesheets" that look like eXtensible Stylesheet Language (XSL).

Comparison

<table>
<thead>
<tr>
<th>XSL</th>
<th>XSLT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output is HTML</td>
<td>Output is XML</td>
</tr>
<tr>
<td>Run in browser/client</td>
<td>Run on server</td>
</tr>
<tr>
<td>Purport: typesetting</td>
<td>Purport: adaptation</td>
</tr>
</tbody>
</table>
An XML input file:

```xml
<?xml version="1.0"?>
<?xml-stylesheet href="gut.xsl" type="text/xsl"?>
<document>
<title>The Underground City</title>
<author>Jules Verne</author>
<date>June 1988</date>
<preface>
<p>
The Goal of Project Gutenberg is to Give Away One Trillion Etext Files by the December 31, 2001. \[10,000 \times 100,000,000 = \text{Trillion}\] This is ten thousand titles each to one hundred million readers, which is only 10% of the present number of computer users. 2001 should have at least twice as many computer users as that, so it will require us reaching less than 5% of the users in 2001.
</p></preface>

<p>We need your donations more than ever!
For these and other matters, please mail to:
</p>

<address>
<name>Project Gutenberg</name>
<street>P. O. Box 2782</street>
<city>Champaign</city>
<state>IL</state>
<zipcode>61825</zipcode>
</address>
</document>
```

- The line
  ```xml
  <?xml version="1.0"?>
  ```
  indicates that this is an XML document.
- The line:
  ```xml
  <?xml-stylesheet href="gut.xsl" type="text/xsl"?>
  ```
says to format this according to an external stylesheet gut.xsl.
An XSL stylesheet

```
<?xml version="1.0" encoding="iso-8859-1" Notícias
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <!-- match the top level document, and print an appropriate heading -->
  <xsl:template match="document">
    <html><head><title><xsl:value-of select="title"/></title></head>
    <body>
      <h1 align="center"><xsl:value-of select="title"/></h1>
      <table align="center">
        <tr><td><xsl:text>Author: </xsl:text><xsl:value-of select="author"/></td></tr>
        <tr><td><xsl:text>Date: </xsl:text><xsl:value-of select="date"/></td></tr>
        <tr><td><xsl:text>Publisher: </xsl:text><xsl:value-of select="publisher"/></td></tr>
      </table>
      <xsl:apply-templates/>
    </body>
  </xsl:template>

  <!-- ignore these fields when they come up -->
  <xsl:template match="title"> </xsl:template>
  <xsl:template match="author"> </xsl:template>
  <xsl:template match="date"> </xsl:template>
  <xsl:template match="publisher"> </xsl:template>

  <!-- include the preface verbatim, with an extra heading -->
  <xsl:template match="preface">
    <h1><xsl:text>Preface</xsl:text></h1>
    <xsl:apply-templates/>
  </xsl:template>

  <!-- put the address into a table, whenever it appears -->
  <xsl:template match="address">
    <table>
      <tr><td><xsl:value-of select="name"/></td></tr>
      <tr><td><xsl:value-of select="street"/></td></tr>
      <tr><td><xsl:value-of select="city"/>
          <xsl:value-of select="state"/>
          <xsl:value-of select="zipcode"/></td></tr>
    </table>
  </xsl:template>

  <xsl:template match="p">
    <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```
The result of this formatting is equivalent to the following HTML:

```
<html>
<head><title>The Underground City</title></head>
<body>
<h1 align="center">The Underground City</h1>
<table align="center">
  <tr><td>Author: Jules Verne</td></tr>
  <tr><td>Date: June 1988</td></tr>
  <tr><td>Publisher: Project Gutenberg</td></tr>
</table>
<h1>Preface</h1>
<p>The Goal of Project Gutenberg is to Give Away One Trillion Etext Files by the December 31, 2001. [10,000 x 100,000,000=Trillion] This is ten thousand titles each to one hundred million readers, which is only 10% of the present number of computer users. 2001 should have at least twice as many computer users as that, so it will require us reaching less than 5% of the users in 2001.</p>
<p>We need your donations more than ever! For these and other matters, please mail to:</p>
<table>
  <tr><td>Project Gutenberg</td></tr>
  <tr><td>P. O. Box 2782</td></tr>
  <tr><td>Champaign, IL, 61825</td></tr>
</table>
</body>
</html>
```
The Goal of Project Gutenberg is to Give Away One Trillion Etext Files by the December 31, 2001. [10,000 x 100,000,000 = Trillion] This is ten thousand titles each to one hundred million readers, which is only 10% of the present number of computer users. 2001 should have at least twice as many computer users as that, so it will require us reaching less than 5% of the users in 2001.

We need your donations more than ever! For these and other matters, please mail to:
What's the point?

- The XSL sheet turns *semantics* into *appearance*.
- It shuffles content around till it makes sense together.
- It creates a final document based upon the original but possibly greatly differing in form.
Deconstructing XSL

- Let's go through the XSL file and try to understand its parts.
- The line:

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
```

says that this is an XML document (like gut.xml) (The extra encoding describes the file's character set, and is optional).
- The line:

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

begins a stylesheet. The specifications after the xml:stylesheet identify the form of the document to follow by referring to specific standards. These are required.
- Within the stylesheet, four directives do most of the work:
  - `xsl:text` delimits literal text to embed in the final document.
  - `xsl:template` describes what to do in a document when you encounter a specific XML tag.
  - `xsl:value-of` allows you to intersperse the text value inside a specific XML tag directly into the results.
  - `xsl:apply-templates` is a special tag that tells the XSL processor to *recurse*. This causes the *content* of the tag being matched to be parsed and matched against templates itself.
**XSL processing**

- An XSL transformation is processed in a very simple fashion.
- First we form the document tree for the source XML document.
- Then we do a *recursive tree walk* through the document tree, from top to bottom and left to right, checking for matches with templates as we go.
- When a template matches a tag in the document tree, we *replace* the tag it matches (and its content) with the content of the template.
- When a tag doesn't match a template, we include its literal content.
- The result is an HTML document based upon the original XML.
**xsl:text**

- While one can *get away* with being sloppy to some extent, all text listed in an xsl stylesheet should be enclosed in

  `<xsl:text>...</xsl:text>`.

- Use this when you wish for text in the document not to be parsed by the XML parser. This includes special HTML directives such as `&lt;`, `&gt;` etc.
- Impact of *not* doing this with normal text is subtle; whitespace in text between xsl tags gets lost. The code:

  ```xml
  <xsl:value-of select="foo"/> <xsl:value-of select="bar"/>
  ```

  does *not* embed a space between the two values. For this, one needs
  ```xml
  <xsl:value-of select="foo"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="bar"/>
  ```

  The effect of the xsl:text directives is to prevent the space from being deleted.
**xsl:template**

- declares an action to take when a tag is matched.
- `match="tag"` describes the tag to match. `match="document"` matches all tags in the source document that are named `document`.
- the content of the `xsl:template` tag *replaces* the content of the matched tag, wherever it appears in the source document.
- `<xsl:apply-templates/>` directive causes processing of tags in the *content* of the matched tag. By default, these tags are *not* processed.
**xsl:value-of**

- It is often handy to be able to insert text from another part of the XML document into a specific place.
- The `xsl:value-of` directive causes the *text content* of a specified tag to be inserted at a particular position within the document. Tag subcontent is ignored.
- The select attribute is a pattern match just like the one for `xsl:template`.
- By default, pattern matches are *relative to the current tag*. For example, if we have the document:

```xml
<?xml version="1.0" ?>
<?xml-stylesheet href="cats.xsl" type="text/xsl"?>
<cats>
  <feline>
    <name>Marcy</name>
    <color>black</color>
  </feline>
  <feline>
    <name>Joe</name>
    <color>calico</color>
  </feline>
</cats>
```

and we are inside `<xsl-template select="feline">`, then `<xsl:value-of select="name"/>` is replaced with the content of the name tag for the feline in question. Let's make a table of all the felines:

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="cats">
    <html><head><title><xsl:text>Cats I know</xsl:text></title></head>
    <body>
      <h1><xsl:text>Cats I know</xsl:text></h1>
      <table border="1">
        <tr><th><xsl:text>Name</xsl:text></th><th><xsl:text>Color</xsl:text></th></tr>
        <xsl:apply-templates/>
      </table>
    </body>
  </xsl:template>
</xsl:stylesheet>
```
This results in the HTML:

<html>
  <head>
    <title>Cats I know</title>
  </head>
  <body>
    <h1>Cats I know</h1>
    <table border="1">
      <tr><th>Name</th><th>Color</th></tr>
      <tr><td>Marcy</td><td>black</td></tr>
      <tr><td>Joe</td><td>calico</td></tr>
    </table>
  </body>
</html>
xsl:apply-templates

- A *recursive* application of the matching rules.
- One template rule should match the whole XML document.
- It "delegates" pattern matching to subsets of the document using xsl:apply-templates.
- You can apply templates to only a subset by including a select attribute.
Example: write an adapter between concepts of Lat, Long

Write an adapter between Latitude=>Lat, Longitude=>Long

Existing input:
<?xml?>
<request>
<lat>42.1</lat>
<long>77.2</long>
</request>

Desired output:
<?xml?>
<getFeatures>
<latitude>42.1</latitude>
<longitude>77.2</longitude>
</getFeatures>

Stylesheet:

<xsl:template match='request'>
   <getFeatures>
      <xsl:apply-templates select='.'/>
   </getFeatures>
</xsl:template>

<xsl:template match='lat'>
   <latitude>
      <xsl:value-of select='.'/>
   </latitude>
</xsl:template>

<xsl:template match='long'>
   <longitude>
      <xsl:value-of select='.'/>
   </longitude>
</xsl:template>
<longitude>
  <xsl:value-of select='.'/>
</longitude>
</xsl:template>
In general

select and match take arbitrary XPATHs! very complex transformations are possible, e.g., adding cross-referencing.

Full programming language capabilities:

xsl:if test=XPATH: conditional inclusion.

xsl:for-each select=XPATH: looping.

...
Can use XSLT to
Make one service seem to be another.
Translate between input and service synonyms.
This is the adapter pattern.
a very simple script to embody a rather complex transform.