Text Encoding Fundamentals and their Application

Constance Crompton
Lee Zickel
Emily C Murphy
Welcome to DHSI 2018!

Thanks for joining the DHSI community!

In this booklet, you will find essential course materials prefaced by some useful information about getting settled initially at UVic, finding your way around, getting logged in to our network (after you’ve registered the day before our courses begin), and so on.

Given our community’s focus on things computational, it will be a surprise to no one that we might expect additional information online for some of the classes - your instructors will let you know - or that the most current version of all DHSI-related information may be found on our website at dhsi.org.

To access the DHSI wifi network, simply go into your wireless settings and connect to the “DHSI” network and enter the password “dhsi2018”.

And please don’t hesitate to be in touch with us at institut@uvic.ca or via Twitter at @AlyssaA_DHSI or @DHInstitute if we can be of any help ....
Psst: Some Suggested Outings

Suggested Outing 1, Botanical Beach (self-organised; car needed)
A self-guided visit to the wet, wild west coast tidal shelf (and historically-significant former research site) at Botanical Beach; we recommend departing early (around 8.00 am) to catch low tide for a better view of the wonderful undersea life! Consider bringing a packed lunch to nibble-on while looking at the crashing waves when there, and then have an afternoon drink enjoying the view from the deck of the Port Renfrew Hotel.

Suggested Outing 2, Butchart Gardens (self-organised)
A shorter journey to the resplendently beautiful Butchart Gardens and, if you like, followed by (ahem) a few minutes at the nearby Church and State Winery, in the Saanich Peninsula. About an hour there by public bus from UVic, or 30 minutes by car.

Suggested Outing 3, Saltspring Island (self-organised; a full day, car/bus + ferry combo)
Why not take a day to explore and celebrate the funky, laid back, Canadian gulf island lifestyle on Saltspring Island. Ferry departs regularly from the Schwartz Bay ferry terminal, which is about one hour by bus / 30 minutes by car from UVic. You may decide to stay on forever....

Suggested Outing 4, Paddling Victoria's Inner Harbour (self-organised)
A shorter time, seeing Victoria’s beautiful city centre from the waterways that initially inspired its foundation. A great choice if the day is sunny and warm. Canoes, kayaks, and paddle boards are readily rented from Ocean River Adventures and conveniently launched from right behind the store. Very chill.

And more!
Self-organised High Tea at the Empress Hotel, scooter rentals, visit to the Royal BC Museum, darts at Christies Carriage House, a hangry breakfast at a local diner, whale watching, kayaking, brew pub sampling (at Spinnaker's, Swans, Moon Under Water, and beyond!), paddle-boarding, a tour of used bookstores, and more have also been suggested!

9:00 to 4:00

Early Class Meeting: 4. [Foundations] DH For Department Chairs and Deans (Hickman 120, Classroom)
Further details are available from instructors in mid May to those registered in the class. Registration materials will be available in the classroom.

3:00 to 5:00

DHSI Registration (MacLaurin Building, Room A100)
After registration, many will wander to Cadboro Bay and the pub at Smuggler’s Cove OR the other direction to Shelbourne Plaza and Maude Hunter's Pub OR even into the city for a nice meal.

Monday, 4 June 2018

Your hosts for the week are Alyssa Arbuckle, Ray Siemens, and Dan Sondheim.

7:45 to 8:15

Last-minute Registration (MacLaurin Building, Room A100)

8:30 to 10:00

Welcome, Orientation, and Instructor Overview (MacLaurin A144)
10:15 to Noon

Lunch break / Unconference Coordination Session (MacLaurin A144, Lab)
(Grab a sandwich and come on down!)

12:15 to 1:15

Undergraduate Meet-up, Brown-Bag (details via email)

1:30 to 4:00

Classes in Session

Institute Panel: Perspectives on DH (or, #myDHis …)
Chair: Alyssa Arbuckle (U Victoria) (MacLaurin A144)

- Milena Radzikowska (Mt Royal C): "Release the Kraken: Story-Driven Prototyping for the Digital Humanities."
  Abstract: I have spent the last 15 years of my career designing text analysis tools for use by humanities scholars. In this brief presentation, I propose to share a concept-based approach to interface design for DH.

- Emily Murphy (U Victoria): "#MyDHIs Edgy."
  Abstract: I will build upon—or, possibly, perform a misprision of—a tweet by Polina Vinogradova; "#myDHIs messy, dusty, edgy, and radically inclusive!" Vinogradova evokes the mess and dust of the archives, the edges that connect nodes of a network, and the political impetus to think of cultural history and community together. I argue that these aspects of DH have a renewed importance as we head into a moment of feminist historiography.

- Margaret Konkol (Old Dominion U): "Prototyping Mina Loy’s Alphabet with a 3D Printer."
  Abstract: This talk discusses the interpretive and methodological implications of using 3D printing technologies to prototype the archival diagrams of a proposed but never constructed plastic segmental alphabet letter kit—a game designed by modernist poet Mina Loy for F.A.O. Schwarz. Although intended as a toy for young children, "The Alphabet that Builds Itself," as a work of "object typography" articulates a theory of language as kinetic, geometric, recombinant, and open to mutation. Alphabetic segments extend into the x, y, and z coordinates in exponential iterations and conjoin with magnets. Combining elements of contemporaneous typefaces like Futura and Gill Sans, which represented modernity’s functional ideals and democratic principles of simplicity, these recombinant letters represent, as this talk argues, Loy’s unpublished modernist poem, an articulation of Loy’s concept of language as a physical fact in which substance, not just form, is semantic.

- Lee Zickel (Case Western Reserve U): "Comfortably Trepid."
  Abstract: #myDHIs found outside the well-established, DH-friendly institutions, at an institution that is devoted predominantly to Medicine and Engineering. I, and with increasing frequency other DH practitioners and instructors, am not positioned in a DH Lab or Humanities Center, but in ITS. Part teacher, part technologist, part translator, I will briefly discuss my work supporting humans and social scientists, particularly those who are new to or less comfortable with computational methodologies.

- Dorothy Kim (Vassar C): "#MyDHIs Antifascist."
  Abstract: I’ve spent a lot of time in the last 12 months thinking about fascism, digital humanities, its long histories, and what it means to do DH work that centers social justice particularly in this global rise of late fascism. I will speak briefly about DH’s history, including the medieval history related to Busa but how that history really connects to data systems that created the Holocaust and also participated in the Cold War nuclear military complex.
Randa El Khatib (U Victoria): "Learning from the Iterative Process."
Abstract: #MyDHis Iterative. In addition to the improvements that come with iterative projects, the iterative process itself is a fruitful area for scholarly inquiry. Within this iterative context, the various teams that I work with and I have been reflecting on and rethinking central DH practices, such as what it means to collaborate, prototype, remix, and implement DH values in our work. In this talk, I will present the various lessons learnt along the way.

Sarah Melton (Boston C): "#MyDHis...People."
Abstract: Taking seriously Miriam Posner’s exhortation to “commit to DH people, not DH projects,” I invite us to reflect on how people are the core of DH. In this brief talk, I will explore the intersections between DH, labor, and infrastructure.

5:00 to 6:00
Opening Reception (University Club)
We are grateful to Gale Cengage for its sponsorship.

Tuesday, 5 June 2018

9:00 to Noon
Classes in Session

12:15 to 1:15
Lunch break / Unconference
"Mystery" Lunches
▼ DHSI Lunchtime Workshop Session (click for workshop details and free registration for DHSI participants)
☆ 73. Introduction to ORCID (Digital Scholarship Commons, Classroom).

1:30 to 4:00
Classes in Session

▼ DHSI Colloquium Lightning Talk Session 1 (MacLaurin A144)
Chair: James O’Sullivan
• New Modes of DH and Archival Skills Acquisition in a Graduate Public History Course. Paulina Rousseau (Ryerson U)
• Walking a Transect: Exploring a Soundscape. John Barber (Washington State U)
• Centering the Edge Case: Designing Services for Humanities Data Research. Grace Afsari-Mamagani (New York U)
• Orwellian Vocabulary and the 21st-Century Politics. Ilgin Kizilgunesler (U Manitoba)
• Making Open Data from a Gray Archive. Sara Palmer (Emory U)

4:15 to 5:15
DHSI Colloquium Lightning Talk Session 2 (MacLaurin A144)
Chair: James O’Sullivan
• Reflections on Founding DHSI. Peter Tandy (U Kansas)
• The Politics of Out-of-Fabrics. John Urry (Durham U)
• The Thematic Analysis of a Postgraduate Digital Humanities Degree. Roman Midden (U Cambridge)
• Reflections on the First Year of DHSI. Steve Linton (U Victoria)

6:00 to 8:00
DHSI Newcomer's Beer-B-Q (Felicitas, Student Union Building)

Wednesday, 6 June 2018

9:00 to Noon
Classes in Session

Lunch break / Unconference
"Mystery" Lunches
▼ Brown Bag Lecture: Alexandra Branzan Albu (U Victoria): "Visual Recognition of Symbolic and Natural Patterns" (Digital Scholarship Commons, 3rd Floor McPherson Library)

Abstract: Image-based object recognition is a visual pattern recognition problem; one may characterize visual patterns as either symbolic or natural. Symbolic patterns evolved for human communication; they include but are not limited to text, forms, tables, graphics, engineering drawings etc. Symbolic patterns vary widely in terms of size, style, language, alphabet and fonts; however, literate humans can easily compensate for this variability and instantly recognize most symbolic patterns. On the other hand, natural patterns characterize images of physical structures; they often lack the intrinsic discriminability and structure of symbolic patterns, and vary widely in terms of pose, perspective, and lighting.

This lecture will explore similarities and differences in approaches designed for recognizing visual and symbolic patterns, and will address the following questions via examples.
- What are the distinctive characteristics of natural patterns? What dimensions of variability can we infer?
- What are the distinctive characteristics of symbolic patterns? What dimensions of variability can we infer?

Alexandra Branzan Albu is an Associate Professor with the Department of Electrical and Computer Engineering and cross-listed with Computer Science. Her research interests are related to image analysis, computer vision, and visual computing. She is actively pursuing outreach activities dedicated to increasing the women's presence in electrical engineering and computer science.

12:15 to 1:15
Classes in Session

1:30 to 4:00
Classes in Session
Thursday, 7 June 2018

9:00 to Noon  Classes in Session

12:15 to 1:15  UVIC Library/ETCL lunchtime talk: “A Humanities Application of 3D printing and Machine Translation in the ChessBard and Loss Sets” by Dr. Aaron Tucker
Digital Scholarship Commons, 3rd floor, Mearns Centre for Learning / McPherson Library
Bring your lunch and come on up!)

1:30 to 4:00  Classes in Session

4:15 to 5:15  DHSI Colloquium Lightning Talk Session 3 (MacLaurin A144)
Chair: James O'Sullivan
- Documenting Deportation: A Collaborative Digital Collection. Paulina Rousseau (Ryerson U)
- Unleashing the Power of Texts as Networks: Visualizing the Scholastic Commentaries and Texts Archive. Jeffrey Witt (Loyola U Maryland) and Drew Winget (Stanford U)
- #haunteDH: Punching holes in the International Busa Machine Narrative. Arun Jacob (McMaster U)
- Text in World: Computational Analysis of Trauma in Genocide Narratives. Nanditha Narayananmooorthy (U York) and Krish Perumal (U Toronto)

7:30 to 9:30  (Groovy?) Movie Night (MacLaurin A144)

Friday, 8 June 2018 [DHSI; DLFxDHSI Opening]

9:00 to Noon  DHSI Classes in Session

12:15 to 1:15  DHSI Lunch Reception / Course E-Exhibits (MacLaurin A100)

1:00 to 2:00  DLFxDHSI Registration (MacLaurin A100)

1:30 to 1:50  [DHSI] Remarks, A Week in Review (MacLaurin A144)

2:00 to 3:00  Joint Institute Lecture (DHSI and DLFxDHSI):
Bethany Nowviskie (CLIR DLF and U Virginia): “Reconstitute the World: Machine-reading Archives of Mass Extinction”
Chair: Lisa Goddard (U Victoria) (MacLaurin A144)

Abstract: The basic constitution of our digital collections becomes vastly more important in the face of two understandings: first, that archives of modernity are archives of the sixth great mass extinction of life on our planet; and next, that we no longer steward cultural heritage for human readers alone. In the same way that we people are shaped by what we read, hear, and see, the machine readers that follow us into and perhaps beyond the Anthropocene have begun to learn from “unsupervised” encounters with our digital libraries. What will we preserve for the living generations and artificial intelligences that will come? What do we neglect, or even choose to extinguish? And from an elegiac archive, a library of endings, can we create forward-looking, speculative collections—collections from which to deep-dream new futures? The most extra/ordinary power we possess is the power to make poetry from records of the past. Could it be called on, one day, to reconstitute the world?
3:30 to 5:00

Joint Reception: DHSI and DLFxDHSI (University Club)
DLFxDHSI Poster/Demo Session
- DHSI Colloquium Poster/Demo Session
  - Media as a Colonialist Artifact in Menzies' Journal. Paula Johanson (U Victoria)
  - Camp Edit: the Institute for the Editing of Historical Documents. Nikolaus Wasmoen (Association for Documentary Editing, U Buffalo), Jennifer Stertz (Association for Documentary Editing, U Virginia), and Cathy Moran Hajo (Association for Documentary Editing, Ramapo C)
  - A Digital Archaeology of Life in Cleveland's Depression-Era Slums. Charlie Harper (Case Western Reserve U) and Jared Benda (Case Western Reserve U)
  - Feminist Pest Control: controlling and not controlling nonhuman pests. Lindsay Garcia (C of William and Mary)
  - Legends of the Buddhist Saints. Jonathan S. Walters (Whitman C) and Dana Johnson (Freelance Web Developer)
  - Accessibility in Digital Environments Via TEI-Encoded Uncontracted Braille. Gia Alexander (Texas A&M U)
  - Translation3point0: Why Literary Translation Data Matters. Katie King (U Washington)
  - PoéticaSonora: A Digital Audio Repository Prototype for Latin American Sound Art and Poetry. Aurelio Meza (Concordia U)
  - Beauty and the Book: Pre-Raphaelite Artistic Practice Contained. Josie Greenhill (U Victoria)
  - Poetic Procedures/Digital Deformances. Corey Sparks (California State U, Chico)
  - Miranda, the Folger Shakespeare Library's new Digital Asset Platform. Meaghan Brown (Folger Shakespeare Library)
  - Living Song Project. Quinn Patrick Ankrum (U Cincinnati) and Elizabeth Avery (U Oklahoma)
  - Digital Frankenstein Variorum. Rikk Mulligan (Carnegie Mellon U)

8:30 to 9:00

DLFxDHSI Registration (MacLaurin A100)

9:00 to 5:30

DLFxDHSI UnConference Sessions
- DHSI All Day Workshop Session (click for workshop details and free registration for DHSI participants)

9:00 to 4:00

- 53. Building Your Academic Digital Identity (MacLaurin D105, Classroom)

Saturday, 9 June 2018 [DLFxDHSI + DHSI Conference and Colloquium]

8:30 to 9:00

- DLFxDHSI Registration (MacLaurin A100)

9:00 to 5:30

- DHSI All Day Workshop Session (click for workshop details and free registration for DHSI participants)

9:00 to 4:00

- 53. Building Your Academic Digital Identity (MacLaurin D105, Classroom)

Welcome

People I: Documenting Online Lives. Chair: Molly Nebiolo (University of New York)

  - Examining Gendered Harassment Online and in Silicon Valley. Andrea Flores (Utica College)
  - This is Just to Say I Have <X> the <Y> in your <Z>: Modernist Memes in an Era of Public Apology. Shawna Ross (Texas A&M University)

Break

People II: Documenting Lives Online. Chair: Dheepa Sundaram (College of Wooster)

  - Youtube Yoga and Ritual on Demand: The Virtual Economics of Hindu Soteriology. Dheepa Sundaram (College of Wooster)
  - The Resemblage Project: Creativity and Digital Health Humanities in Canada. Andrea Charise (University of Toronto) and Stefan Krecsy (University of Toronto)

Lunch

Projects I: Building and Analyzing. Chair: Yannis Rammos (New York University)

  - Building the ARTECHNE Database: New directions in Digital Art History. Marrieke Hendriksen (Old Dominion University)
  - The Ineffective Inquisition: The Holy Office’s Sphere of Influence in Early Modern New Spain. Kira Homo (Pennsylvania State University)

Break

Projects II: Mapping and Visualizing. Chair: Innocent Opara (Qumet Institute)

  - Mapping Sarah Sophia Bank’s Numismatic Collection. Erica Hayes (North Carolina State University) and Kacie Wills (University of California, Riverside)
  - Text Mining and Visualizing 18th Century American Correspondence. Ashley Sanders Garcia (University of California, Los Angeles)

Break

Practices: Digital Scholarship on Campus and in the Classroom. Chair: Alyssa Arhuckle (University of Victoria)
Concluding Remarks

Sunday, 10 June 2018 [SINM + DHSI Registration, Workshops]

8:30 to 9:00  Symposium on Indigenous New Media Registration (MacLaurin A100)

9:00 to 5:00  DHSI Registration (MacLaurin A100)

▼ SINM Sessions

9:00 to 4:00  
- 63. Symposium on Indigenous New Media: Reading Group (Hickman 105, Classroom)
- 72. Symposium on Indigenous New Media: Indigitization (Hickman 120, Classroom)
  Full details here

▼ DHSI All Day Workshop Sessions  (click for workshop details and free registration for DHSI participants)

9:00 to 4:00  
- 53. Building Your Academic Digital Identity (MacLaurin D105, Classroom)
- 54. An Introduction to the Archaeology of 1980s Computing (MacLaurin D114, Classroom)

▼ DHSI AM Workshop Sessions  (click for workshop details and free registration for DHSI participants)

9:00 to Noon  
- 55. Regular Expressions (MacLaurin D111, Classroom)
- 56. 3D Visualization for the Humanities (MacLaurin D010, Classroom)
- 58. DH Fieldwork Methods (MacLaurin D016, Classroom)
- 60. Pedagogy of the Digitally Oppressed: Inculcating De-/Anti-/Post-Colonial Digital Humanities (MacLaurin D107, Classroom)
- 61. Introduction to #GraphPoem. Digital Tools for Poetry Computational Analysis and Graph Theory Apps in Poetry (MacLaurin D101, Classroom)
- 62. Creating a CV for Digital Humanities Makers (MacLaurin D115, Classroom)

▼ DHSI PM Workshop Sessions  (click for workshop details and free registration for DHSI participants)

1:00 to 4:00  
- 64. Agent-Based Modelling in the Humanities (MacLaurin D111, Classroom)
- 65. Unleash Linux on MacOS (MacLaurin D010, Classroom)
- 66. DHSI Knits: History of Textiles and Technology (MacLaurin D016, Classroom)
- 67. Crowdsourcing as a Tool for Research and Public Engagement (MacLaurin D109, Classroom)
- 69. Web Annotation as Critical Humanities Practice (MacLaurin D103, Classroom)
- 70. Dynamic Ontologies for the Humanities (MacLaurin D107, Classroom)
- 71. Social Media Research in the Humanities (MacLaurin D101, Classroom)

▼ Joint Institute Lecture (DHSI and SINM):  
David Gaertner (U British Columbia): "A Landless Territory?: CyberPowWow and the Politics of Indigenous New Media."  
Chair: Deanna Reder (Simon Fraser U)  
(MacLaurin A144)

Abstract: Following the 1997 launch of Skawennati’s (Mohawk) CyberPowWow, digital space has become a vital new territory for the resurgence of Indigenous storytelling and cultural practice: "We have signed a new treaty," Cree artist Archer Pechawis wrote of this period, "and it is good. We have the right to hunt, fish, dance and make art at www.CyberPowWow.net, .org and .com for as long as the grass grows and the rivers flow." This talk will critically explore the theoretical, cultural, political-economic, and gendered dynamics underwriting the histories and futures of Indigenous new media. Particular attention will be given in examining the ways in which new media and digital storytelling connect to and support key issues in the field of Indigenous studies, such as sovereignty, self-determination, decolonization, and land rights.

4:10 to 5:00

After the day, many will wander to Cadboro Bay and the pub at Smuggler's Cove OR the other direction to Shelbourne Plaza and Maude Hunter's Pub OR even into the city for a bite to eat.

Monday, 11 June 2018 [DHSI + SINM]
Your hosts for the week are Ray Siemens and Dan Sondheim.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:45 to 8:15</td>
<td>DHSI Last-minute Registration (MacLaurin A100)</td>
</tr>
<tr>
<td>8:30 to 10:00</td>
<td>DHSI Welcome, Orientation, and Instructor Overview (MacLaurin A144)</td>
</tr>
<tr>
<td>9:00 to 4:00</td>
<td>SINM Sessions</td>
</tr>
</tbody>
</table>

- DHSI Classes in Session (click for details and locations)
  - 29. [Foundations] Models for DH at Liberal Arts Colleges (& 4 yr Institutions) (MacLaurin D109, Classroom)
  - 32. Stylometry with R: Computer-Assisted Analysis of Literary Texts (Clearihue A102, Lab)
  - 33. Digital Storytelling (MacLaurin D111, Classroom)
  - 34. Text Mapping as Modelling (Clearihue D131, Classroom)
  - 35. Geographical Information Systems in the Digital Humanities (Clearihue A105, Lab)
  - 36. Open Access and Open Social Scholarship (MacLaurin D114, Classroom)
  - 37. Introduction to Machine Learning in the Digital Humanities (Cornett A229, Classroom)
  - 38. Queer Digital Humanities: Intersections, Interrogations, Iterations (MacLaurin D110, Classroom)
  - 41. Using Fedora Commons / Islandora (Human and Social Development A160, Lab)
  - 42. Documenting Born Digital Creative and Scholarly Works for Access and Preservation (MacLaurin D115, Classroom)
  - 43. Games for Digital Humanists (MacLaurin D016, Classroom & Human and Social Development A170, Lab)
  - 44. XPath for Document Archeology and Project Management (Cornett A128, Classroom)
  - 46. Surveillance and the Digital Humanities (MacLaurin D103, Classroom)
  - 47. Text Analysis with Python and the Natural Language Toolkit (Clearihue A103, Lab)
  - 49. Wrangling Big Data for DH (Human and Social Development A150, Lab)
  - 50. Accessibility & Digital Environments (MacLaurin D101, Classroom)
  - 51. Critical Pedagogy and Digital Praxis in the Humanities (MacLaurin D105, Classroom)
  - 52. Drupal for Digital Humanities Projects (MacLaurin D107, Classroom)

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:15 to Noon</td>
<td>DHSI Classes in Session (click for details and locations)</td>
</tr>
<tr>
<td>12:15 to 1:15</td>
<td>DHSI Undergraduate Meet-up, Brown-Bag (details via email)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>DHSI Classes in Session</td>
</tr>
</tbody>
</table>

- Joint Institute Lecture (DHSI and SINM):
  - Jordan Abel (Simon Fraser U): “Indigeneity, Conceptualism, and the Borders of DH.”
  - Chair: Michelle Brown (U Hawaii) (MacLaurin A144)

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:10 to 5:00</td>
<td>Joint Reception: DHSI and SINM (University Club)</td>
</tr>
</tbody>
</table>

Tuesday, 12 June 2018

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00 to Noon</td>
<td>Classes in Session</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:15 to 1:15</td>
<td>DHSI Lunchtime Workshop Session (click for workshop details and free registration for DHSI participants)</td>
</tr>
</tbody>
</table>

- 73. Introduction to ORCID (Digital Scholarship Commons, Classroom).
<table>
<thead>
<tr>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:30 to 4:00</td>
<td>Classes in Session</td>
</tr>
</tbody>
</table>
| 4:15 to 5:15 | DHSI Colloquium Lightning Talk Session 4  
|              | Chair: Lindsey Seatter                                                             |
|              | - Mapping Indigenous and Chicana/o Environmental Imaginaries using GIS. Stevie Ruiz (California State U, Northridge), Quetzalli Enrique (California State U, Northridge), Enrique Ramirez (California State U, Northridge), and Tomas Figueroa (California State U, Northridge)  
|              | - Doing DH with Graphic Narratives. John Barber (Washington State U)               |
|              | - “But is it any good?”: A quantitative approach to the popularity of digital fanfiction. Suzanne Black (U Edinburgh)  
|              | - The American Prison Writing Archive (APWA). Doran Larson (Hamilton C), Janet Simons (Digital Humanities Initiative, Hamilton C), and William Rasenberger (Hamilton C) |
| 6:00 to 8:00 | DHSI Newcomer's Beer-B-Q (Felicitas, Student Union Building)                        |
| 9:00 to Noon | Classes in Session                                                                 |
| 12:15 to 1:15| Lunch break / Unconference                                                           |
|              | “Mystery” Lunches                                                                  |
| 1:30 to 4:00 | Classes in Session                                                                 |
| 4:15 to 5:15 | DHSI Colloquium Lightning Talk Session 5  
|              | Chair: Lindsey Seatter                                                             |
|              | - Faraway, so close: Has the political environment really changed in Ecuador?. Luis Meneses (Electronic Textual Cultures Lab, U Victoria)  
|              | - Re-mixing Melville's Reading: Text Analysis of Marginalia with R and XSLT. Christopher Ohge (U London, School of Advanced Study) and Steven Olsen-Smith (Boise State U)  
|              | - Developing Interactive and Open-Source OER: Inquiry-Based Music Theory. Evan Williamson (U Idaho)  
|              | - Spatial Humanities and the Web of Everywhere. Ken Cooper (SUNY Geneseo)           |
| 6:00 to 7:00 | “Half Way There (yet again)!” [An Informal, Self-Organized Birds of a Feather Get-Together] (Felicitas, Student Union Building)  
|              | Bring your DHSI nametag and enjoy your first tipple on us!                         |
| 7:30 to 9:30 | (Groovier?) Movie(r) Night (MacLaurin A144)                                        |
| 9:00 to Noon | Classes in Session                                                                 |
| 12:15 to 1:15| Lunch Reception / Course E-Exhibits (MacLaurin A100)                                |
|              |                                                                                     |
**1:30 to 2:30**

(MacLaurin A144)

Abstract: Much has changed and continues to change in digital humanities since the formal establishment of Iter in the Fall of 1997. However, the mandate of the not-for-profit partnership to support “the advancement of learning in the study and teaching of Middle Ages and Renaissance (400–1700) through the development and distribution of online resources” continues to have relevance. This presentation explores the striking challenges faced by Iter and presents our current thinking on the realization of this mandate for the future through a platform with a focus on facilitating the discovery of the academic resources necessary to our work; creating an environment for collaboration, sharing and developing projects; and on enabling the distribution and publication of our scholarship.

**2:40 to 3:00**

Awards and Bursaries Recognition
Closing, DHSI in Review (MacLaurin A144)

**Contact info:**
institut@uvic.ca  P: 250-472-5401  F: 250-472-5681
Text Encoding Fundamentals
Coursepak for DHSI 2018*

Dr. Constance Crompton, University of British Columbia at Okanagan
Dr. Emily Murphy, Queen’s University
Lee Zickel, Case Western Reserve University

*These materials have been adapted from Julia Flanders, Syd Bauman, and the Women Writers Project’s educational resources for teaching TEI. Additional materials were shared from Kailey Fukushima at the Digital Craik project. We would like to thank them all for their scholarly generosity.

Table of Contents

<table>
<thead>
<tr>
<th>Content</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logistics &amp; Syllabus</td>
<td>2</td>
</tr>
<tr>
<td>Downloading and Installing oXygen</td>
<td>3</td>
</tr>
<tr>
<td>Reading Lists</td>
<td>4</td>
</tr>
<tr>
<td>Project and Document Analysis Considerations</td>
<td>5-7</td>
</tr>
<tr>
<td>Project Considerations (Template)</td>
<td>5-6</td>
</tr>
<tr>
<td>Document Analysis</td>
<td>7</td>
</tr>
<tr>
<td>Elements for Basic TEI Documents</td>
<td>8-15</td>
</tr>
<tr>
<td>Element Groups</td>
<td>8</td>
</tr>
<tr>
<td>Alphabetical List of Elements</td>
<td>9-15</td>
</tr>
<tr>
<td>Alphabetical List of Attributes</td>
<td>14-16</td>
</tr>
<tr>
<td>oXygen Shortcuts</td>
<td>17</td>
</tr>
<tr>
<td>Notes</td>
<td>18</td>
</tr>
<tr>
<td>What is TEI?</td>
<td>19-23</td>
</tr>
<tr>
<td>Getting Started Using the TEI - Ch 1</td>
<td>24-42</td>
</tr>
<tr>
<td>Getting Started Using the TEI - Ch 2 (Optional)</td>
<td>44-58</td>
</tr>
<tr>
<td>An Even Gentler Introduction to XML</td>
<td>59-74</td>
</tr>
<tr>
<td>Text Encoding</td>
<td>75-85</td>
</tr>
</tbody>
</table>
Logistics

Before DHSI:

Please:

- Visit http://tapasproject.org/ and create an account. (see: Joining TAPAS below)
- Be sure you understand your computer's directory structure and where your files are saved when you save them.
- Prepare a (few) selected text(s) to work on during the workshop. This can be a printed text transcribed into plain text (.txt) format, or a pre-digitized text transcribed into plain text (.txt) format. Preferably something complicated enough to be interesting, but not complicated enough to be frustrating. We do have some materials if you need them!
- Once an instructor has contacted you with licensing information download and install oXygen. (see: Downloading and Installing oXygen, below)
  - note: oXygen comes with a 90-day trial. If you are unsure whether you will buy oXygen, make sure your trial doesn't expire during DHSI.
  - note: The course will not be taught in a UVic computer lab so please bring your laptop!

During DHSI:

- Suggested and Supplemental readings have been provided as additional points of access to learning TEI. Neither sets are required, but you may find the Suggested readings useful.
- Excepting some DHSI welcome and closing events, which may alter things, the following will hold:
  - DHSI room location - Cornett A128
  - 9:30am - 12:00pm Morning Session
  - 12:00pm - 1:15pm Lunch
  - 1:15pm - 3:50pm Afternoon Session

Classes start at 10:00 on the Monday and end at 12:00 on the Friday.

Syllabus:


Wednesday: Empty elements, Milestones, Overlap issues, Figures, Manuscript encoding, Interpretive annotation, Case studies.

Thursday: TEI Header, Data modeling, Roma, Schema constraint; TEI Publication, Hands on.

Friday: Next steps, Building a team, TAPAS, XSLT, Advanced TEI, Finding help.
Downloading and Installing oXygen

TEI-XML is software-independent. However, for consistency's sake, we will all use oXygen (XML editing software) in this course. **You will be using our own laptop in this course**, so we recommend installing a trial version of the software on your machine. Alternately, you may purchase a perpetual academic user license at $99.

One of your instructors will be contacting you with a trial license. Any alterations to the installation instructions below will be added to the license information email.

**Trial Version Installation**

1. Go to [http://www.oxygenxml.com/register.html](http://www.oxygenxml.com/register.html) and click on the "Get Trial License" button. Fill out the form, and click on the "Get License" button. Oxygen will send you an email with trial license key (NB: There are three Oxygen products. It is the Oxygen Editor that you want).
2. Go to [http://www.oxygenxml.com/download_oxygenxml_editor.html](http://www.oxygenxml.com/download_oxygenxml_editor.html) and click on the tab that corresponds with your operating system (either Linux (select the 32-bit option), Mac OSX, or Windows (select the 32-bit option)).
3. Move the compressed file that you have downloaded into your Applications (Mac) or Programs (PC) directory. Double click to open it.
4. This should create a directory called oxygen. Inside you will find the program, Oxygen_XML_Editor.app. Open it.
5. Follow the installation instructions that pop up. When you come to a screen with a blank field that asks for your license paste in the nine-line trial license key. An instructor will provide you with this key closer to the date of DHSI.

Joining TAPAS

Thankfully, sign up is pretty straightforward. If you already have a TAPAS account, skip to step ##

1. Go to tapasproject.org and click on "Sign Up."
2. Click the "Create new account" tab.
3. Fill in all the information.
5. If that fails...This will be a really important step: Email Lee (lxz11@case.edu) and tell him your TAPAS username. Once we have your username, we will invite you to the TAPAS project area and collections we'll be using for the course. Please accept the invitations!
Reading List

Suggested Readings

Birnbaum, David. "What is XML and why should humanists care? An even gentler introduction to XML." Digital Humanities at Oxford.

"Chapter 1 - Getting Started Using the TEI." TEI@Oxford.(Chapter 2 has also been included in the Coursepak, but is supplemental.)


Flanders, Julia. "What is TEI?" Women Writers Project

Supplemental Readings


Reference Websites

oXygen Editor Forum http://www.oxygenxml.com/forum/

TEI Archiving Publishing and Access Service (TAPAS) http://www.tapasproject.org/

TEI By Example http://www.teibyexample.org/


Project and Document Analysis Considerations

I. Project Considerations (Template)

When you envisage your completed project what do you see? Whether you are planning your own project, facilitating someone else's project, or working on an existing project, try to outline who the project will serve, what it will enable, and what you will have to produce to see it through.

**Title**
- What is the project's title? Unless obvious, explain how the title relates to the subject of the project.

**Description**
- What is the purpose of the project (e.g. preservation, research, dissemination)?
- What type of project is it (e.g. electronic edition, archive, exhibit, data viz, GIS map, etc.)?
- What content will be included: images (page, photograph, painting, map, music, etc.), tables, data visualization, transcriptions, audio files, etc.?
- What will be the content's scope: breadth (corpora, single work, collection, etc.), timeframe (period, decade, year, etc.)?
- Which methodological or political perspective informs your project?
- What concepts link your documents? Do they belong together because they have common author, time period, geographic location, genre, medium, or other formal characteristics?

**Aims and Rationale**
- What does the project contribute to the field, library, university, or general populace?
- How does it relate to existing projects? Are there competing projects (digital or otherwise)?
- What gap does your project fill?
- What types of research questions does your project enable?
- What will encoding enhance about the interpretation, display, and/or dissemination of your documents?

**Audience**
- Who will be accessing the project (e.g. students (level), scholars (field), alumni, general public) and how will they interact with the project?
- Will a user guide or instructions be necessary?

**Resources**
- What resources will the project need (e.g. hosting, hardware, software, specific digital tools)?
- Do your project's documents exist already, or will you have to write them? Will you have to transcribe them?
• In addition to finding or producing primary documents, which supplemental material (such as introductions, biographies, indexes, or diagrams) will you have to prepare?
• What issues do you anticipate (if any), and how do you plan to address them?

Collaborators
• Who will work on the project? List all collaborators including library and IT staff, and describe their roles.

Funding Allocations
• What are the anticipated expenses for the project: software, hardware, human resources, etc.?
• How will the project be funded? Do you plan to apply for other funding resources?

Copyright and IRB
• Are permissions needed for copyrighted material?

Timeline
• In what form does the data exist (spreadsheet, Word docs, PDFs, mp3s, jpeg/tiff, etc.)?
• What (if any) work has been completed thus far?
• What will your workflow look like? (Declare some steps in the workflow process black boxes for now—you don’t have to learn how to generate everything you need right away.)
• Define the benchmarks: what are the key components of each major stage of the project (minimally--research, production, dissemination, curation), who will be involved in each stage, and what assessment measure will be applied to each stage to determine whether to proceed, revise, or abort?

Dissemination
• Which existing projects or publishing platforms might help you decide the best way to publish your work?
• What scholarship could result from this project (e.g. conference paper, presentation, and/or poster; peer-reviewed article)? Who will be involved in its production?
• Will the project have a social media presence (e.g. on Facebook, Twitter, Tumblr, Google+, dedicated Blog, etc.)? Who will be responsible for it?

Curation
• Will a continuity journal be kept of decisions made throughout the project about matters like software, processes, protocols, metadata, encoding, etc.?
• Have you written a data management plan using the DMP Tool or following the NEH-ODH guidelines?
• Where will the data be stored during each stage of the project?
• Will the data (also) be available to other researchers through a repository?

Adapted from Leigh Bonds, CWRU
II. Document Analysis
Consider the sample document(s) you've chosen to focus on, and imagine a hypothetical audience for your digital version of it. Then make some notes on your document based on the following questions.

Preliminary Questions
- How is the document structured? What are its major structural chunks?
- Within those chunks, what are the key features of the document (images, poetry, dialogue, etc.)?
- What features will the audience need more information about (for instance, special terminology, historical events)? What kind of supplemental information could you supply (e.g. a glossary, regularization of proper names.)?
- Are there any kinds of regularization or editorial amendment you will perform as you transcribe the text? What is the rationale for these changes? What options will they add (or remove) for your audience?
- How much information about the appearance of the document do you need to capture? What is the significance of this information for the reader?
- Are there significant themes or topics that you can identify in these documents?

Mid-Course Questions
Try answering the following questions once you have started marking up your texts.
- How will you encode the major structural chunks in your document?
- How will you encode the key features of the document?
- Will you encode significant themes or topics in these documents? How?
- Which encoding practices will you need to link information or avoid overlap?
- What interface features can you support with the encoding you've planned so far (e.g. in your notes above)?
- Are there any features of your document that TEI doesn't appear to support?
Elements for basic TEI documents

This is more of a brief reference sheet than an exhaustive list of TEI elements: it is intended to provide you with a way to look up the most commonly used elements, grouped together for the exercises in which we'll be encountering them. For detailed information about the contents and semantics of these elements (and for other more arcane elements), have a look at the TEI Guidelines [http://www.tei-c.org/release/doc/tei-p5-doc/en/html/].

Element groups

structure
  <TEI>, <back>, <body>, <front>, <group>, <teiHeader>, and <text>

general purpose block-level
  <ab>, <argument>, <div>, <head>, <item>, <label>, <list>, <p>, <quote>, and <said>

general purpose phrase-level
  <bibl>, <date>, <distinct>, <emph>, <foreign>, <hi>, <mentioned>, <name>, <q>, <quote>, <rs>, <said>, <seg>, <soCalled>, and <term>

poetry
  <l>, <lg>, and <rhyme>

drama
  <castGroup>, <castItem>, <castList>, <role>, <roleDesc>, <sp>, <speaker>, and <stage>

diary entries, letters, etc.
  <closer>, <dateline>, <opener>, <postscript>, <salute>, <signed>, and <trailer>

alternative transcriptions
  <abbr>, <choice>, <corr>, <expan>, <orig>, <reg>, and <sic>

manuscripts and physicality of documents
  <add>, <addSpan>, <cb>, <del>, <delSpan>, <handShift>, <lb>, <milestone>, and <pb>

editorial annotation
  <app>, <damage>, <gap>, <lem>, <rdg>, <restore>, <subj>, <supplied>, and <unclear>

hypertextual
  <anchor>, <note>, <ptr>, and <ref>
Elements (in alphabetical order)

<TEI>
The outermost (or ‘root’) element for any TEI P5 conformant document. It groups together the TEI header and the document text. It must have the TEI namespace specified, and should have an xml:lang=attribute, i.e. <TEI xmlns="http://www.tei-c.org/ns/1.0" xml:lang="en">.

<ab>
An ‘anonymous block’, that is, a paragraph-like chunk that does not carry the semantic weight of a paragraph. Use type=and maybe subtype=to categorize.

<abbr>
An abbreviation; may be used alone or, when inside <choice>, in combination with <expan> which holds an expanded reading.

<add>
A handwritten addition. The hand=attribute indicates the handwriting in which the addition is made. This attribute contains an identifier which points to a <hand>element in the <profileDesc> of the TEI header; this <hand>element contains an extended description of the handwriting, ink, and other details.

<addSpan>
An empty element which marks the starting point for a handwritten addition that either is too long to be encoded with <add>, or overlaps an element boundary. Its spanTo=attribute points to an <anchor>element which marks the endpoint of the added material. The hand=attribute indicates the handwriting in which the addition is made (see above for details).

<anchor>
An anchor point, usually used as a place for some other element (such as a note) to point to, using the anchor’s xml:id=attribute.

<app>
Contains one entry in a critical apparatus, with an optional lemma and at least one reading.

<argument>
A short summary or description of the contents of the following section. Contains one or more <p>or <lg>elements.

<back>
Contains the back matter of the document, if any: indices appendices, epilogues, colophons, errata lists, etc. May be subdivided into <div>elements if necessary.

<bibl>
Used to encode bibliographical references, either in a list (using <listBibl>) or in running prose.

<body>
Contains the main body of the document, not including front matter and back matter. The <body> element typically includes one or more <div>elements. It may start with a <head>. (Think about where the <head> belongs—-is it the heading for the body, or the heading for the first division?)
<castGroup>
A grouping of related items in a cast list, containing one or more <castItem> elements and an optional <head> and <trailer>.
</castGroup>

<castItem>
An item in a cast list, containing a <role> and an optional <roleDesc>.
</castItem>

<castList>
A cast list in a dramatic text, listing the roles in the drama. It consists of one or more <castItem> or <castGroup> elements.
</castList>

<cb>
An empty element which marks the break between one column and the next. Equivalent to <milestone unit="column">.
</cb>

<choice>
Groups together two or more alternate encodings of a phrase-level passage, using the elements listed below.
</choice>

<closer>
Very similar to <opener>, but located at the end of the <div> instead of at the beginning.
</closer>

<corr>
A corrected reading of a typographical error or oddity in the original; may be used alone or, when inside <choice>, in combination with <sic>, which holds the original reading.
</corr>

<damage>
A damaged portion of the original text; the type= attribute allows you to classify the damage, and the extent= attribute allows you to indicate the extent of the damage.
</damage>

<date>
Used to encode dates. The when= attribute can be used to encode a regularized form of the date (e.g. <date when="2001">The first year of the new century</date> or <date when="2005-05-29">Sun, 29 May 05</date>).
</date>

<dateline>
Used within <opener> and <closer> to encode the date and place of writing. Contains words and phrase-level encoding.
</dateline>

<del>
A deletion. The hand= attribute indicates the handwriting in which the addition is made (see above for details).
</del>

<delSpan>
An empty element which marks the starting point for a deletion that is either too long to be encoded with <del> or that overlaps an element boundary. Its spanTo= attribute points to an <anchor> element which marks the endpoint of the deleted material. The hand= attribute indicates the handwriting in which the deletion is made (see above for details).
</delSpan>

<distinct>
Used for linguistically distinct words (e.g. dialect words, regionally accented words).
</distinct>

<div>
A division of a text: for instance, an act, a chapter, a section, a poem, a letter... Use the type= attribute to indicate what kind of division.
</div>
<emph>
  Used to encode emphasized words or phrases.
</emph>

<expan>
The expanded reading of an abbreviation; typically used inside <choice>, in combination with <abbr> which holds the corresponding abbreviated reading. Rarely used alone.
</expan>

<foreign>
  Used for foreign-language words when no other element (e.g. <quote>) is already present.
</foreign>

<front>
  Contains the front matter of the document, if any: title pages, tables of contents, introductory essays, and so forth. The <front> element contains an optional <titlePage> and may be subdivided into <div> elements.
</front>

<gap>
  A gap in the original text (either from damage, deletion, excision, or some other cause). The <desc> child element provides a description of what is missing, and the reason=attribute provides the reason for the omission.
</gap>

<group>
  This element is used to represent documents which contain more than one independent text. It appears instead of <body> in the overall TEI document structure, and groups together multiple <text> elements, with an optional <front> and <back>.
</group>

<handShift>
  An empty element which marks the boundary point at which a change of handwriting takes place. Its new=attribute indicates the handwriting that begins at the point being marked. The new=attribute functions just like the hand=attribute, in pointing to a <hand> element in the TEI header, which provides detailed information on the handwriting in question.
</handShift>

<head>
  The heading of a division: contains words and phrase-level encoding. <head> may appear at the start of <div>, but also at the start of <body>, <front>, <back>, <list>, and <lg>.
</head>

<hi>
  Used to encode words or phrases which are highlighted for reasons which the encoder either does not know or chooses not to analyze.
</hi>

<item>
  An item in a list: contains an optional <label> followed by words and phrase-level encoding, or a series of paragraphs.
</item>

<l>
  A single verse line: contains words and phrase-level elements. May have a met=attribute to formally specify the metrical pattern.
</l>

<label>
  The label of an item (e.g. a letter, number, or word indicating its order or other facts about it): contains words and phrase-level encoding. Note that <label> can also be the first element inside a paragraph.
</label>

<lb>
  An empty element which marks a typographical line break. Equivalent to <milestone unit="line">.
</lb>
A lemma; e.g., the reading from the base text.

A group of verse lines: contains one or more lg elements. May have a rhyme=attribute to formally specify the rhyme scheme, e.g. <lg rhyme="ABAB">.

A list: contains a series of item elements.

Used for words which are mentioned but not used (for instance, for spelling or definition purposes).

An empty element which marks a boundary point in the text according to some standard reference system, such as signatures, scrolls, leaves. Use the unit=attribute to indicate the reference system whose units are being marked at this point.

Used to encode all kinds of names, i.e. proper nouns and noun-phrases. If you want to distinguish between different kinds of names, you can use the type=attribute (e.g. <name type="person">). TEI also includes specific elements for different kinds of names (e.g. <persName>) for projects that need more detailed encoding. The rs element is a more generic version of name, which may be used to encode common nouns and noun phrases.

A note (a footnote, endnote, marginalnote, or linенote). Link the note to the point where it’s anchored using xml:id=and target=. <note> contains most anything, including words and phrase-level encoding, or one or more p elements.

This element may appear at the start of a div, text, front, or back, and it groups together the elements that appear at the start of a letter or similar document: the date and place of writing (using dateLine), and the salutation to the person being addressed (using salute).

An unmodernized reading in the original; may be used alone or, when inside choice, in combination with reg, which holds a regularized reading.

A prose paragraph: contains words and phrase-level encoding.

An empty element which marks the break between one page and another. By convention, information stored in the attributes of pb refer to the page that follows the break. Equivalent to <milestone unit="page">.

Indicates a reference to some other XML element (either in the current document or some other accessible document) by pointing to it with a URI on the target=attribute. Must not have content. E.g., <ptr target="#art08_sec08"/>.
<postscript>
  Used to encode a postscript, e.g. of a letter.
</postscript>

<q>
  Used to encode passages surrounded by quotation marks, when you don’t want to bother with a more precise element like <said>. Roughly the same as <hi rend="surrounded-with-quotation-marks"/>
</q>

<quote>
  Used to encode quotations from other sources; contains words and phrase-level encoding.
</quote>

<rdg>
  A single reading, e.g. from a particular witness.
</rdg>

<ref>
  Indicates a reference to some other XML element (either in the current document or some other accessible document) by pointing to it with a URI on the target=attribute. May (and probably should) have content. E.g., <ref target="#art08_sec08">the <soCalled>IP</soCalled> clause</ref>.
</ref>

<reg>
  A modernization of a reading in the original; may be used alone or, when inside <choice>, in combination with <orig>, which holds the corresponding unmodernized reading.
</reg>

<restore>
  Indicates restoration of text to an earlier state by cancellation of a marking or instruction; in particular, useful to indicate that a deletion was restored, e.g. by the notation "stet".
</restore>

<rhyme>
  May be optionally used to indicate the portion of the metrical line that rhymes, and with its label=attribute which part of the rhyme scheme is in play.
</rhyme>

<role>
  The name of a role in a cast list.
</role>

<roleDesc>
  The description of a role in a cast list.
</roleDesc>

<rs>
  Used to encode all kinds of references to people, places, and things; i.e., nouns and noun phrases. If you want to distinguish between different categories of entity being referred to, you can use the type=attribute (e.g. <rs type="person">). The <name>element is a more specialized version of <rs>, reserved for proper nouns and noun-phrases.
</rs>

<said>
  Passages spoken aloud or thought, e.g. by a character in a novel.
</said>

<seg>
  General-purpose phrase-level segment: use type= and maybe subtype= to categorize.
</seg>

<salute>
  Used within <opener> and <closer> to encode the salutation to the person being addressed (e.g. “Dear Sir”, or “I remain faithfully yours...”). Contain words and phrase-level encoding.
</salute>

<sic>
  A typographical error or oddity in the original; may be used alone or, when inside <choice>, in combination with <corr>, which holds a corrected reading.
</sic>
<signed>
Used within <closer> to encode the signature or name of the person writing. Contains words and phrase-level encoding.

<soCalled>
Used to encode (or express) authorial distance; e.g., phrases that were or should be in scare quotes.

<sp>
A dramatic speech; usually begins with a <speaker> element, followed by a <p> or <lg>.

<speaker>
A speaker identification printed in the text.

<stage>
A stage direction. The type= attribute may be used to identify the kind of stage direction; suggested values include:

- Business
- Costume
- Delivery
- Entrance
- Exit
- Location
- Narrative
- Novelistic

<subst>
Groups together an <add> and a <del> so that the addition is understood as being a substitution for the deletion.

<supplied>
Indicates that a given word or passage cannot be read in the original and is being supplied (either through editorial judgment or from some other textual source).

<teiHeader>
The wrapper for all of the document’s metadata. The elements that go inside the TEI header are too numerous to list usefully here; see the templates for details.

<term>
Used to encode specialized terminology; often associated with a <gloss>.

<text>
The wrapper element which contains all of the document’s content. The <text> element is most often used for a single work (i.e., a single published document, or a single aesthetic unit such as a play or a work of fiction). Terms like “single work” and “aesthetic unit” need to be defined by the individual project. A <text> element contains an optional <front>, a mandatory <body>, and an optional <back>.

<trailer>
This element is used for things that come at the very end of the document or section, such as “The End”.

<unclear>
Indicates that a given word or passage is unclear, but not entirely illegible (expresses uncertainty rather than absolute lack of information); multiple alternative readings may be grouped in a <choice> element.
Attributes (in alphabetic order)

met=
  May be used to specify the metrical pattern a line (or line group).

n=
  Provides a label or identifier for this particular element, not necessarily unique.

next= and prev=
  Allow what is logically a single text object (e.g. a quotation) to be encoded as a series of two or more discrete XML elements, as a work-around for overlap problems. These attributes represent the connections between these fragmentary elements, by pointing to a prior or subsequent element in the chain of fragments. They do so by referring to that element’s xml:id=value.
  That is, if next=is specified on a <said>element, then its value should be a hash mark (#) followed by the value of the xml:id=of another <said>element, the one that is the next part of the spoken passage. For example, <said xml:id="s01" next="#s02">Hey</said>, he said, <said xml:id="s02" prev="#s01">What's up?</said>.

rend=
  May be used to specify how the element looked in the source. E.g.,<head rend="align (center)" or <i rend="indent(2)". 

rhyme=
  May be used to specify the rhyme scheme of a linegroup.

target=
  Provides a URI (e.g.http://bauman.zapto.org/gallery/Niagara_Falls_2008-01/2008_01_07T16_35_39or #sect08) that points to either another document or an element within an XML document (including the current one).

xml:id=
  Provides a unique identifier for this particular element, thus allowing other elements to point to it (using their target=, next=, prev=, etc.).

xml:lang=
  Used to indicate the language of an element’s content. Its value conforms to BCP 47 (a standard system for defining language codes). For information on how BCP 47 codes are constructed, see the note in the data language [http://www.tei-c.org/release/doc/tei-p5-doc/en/html/ ref-data.language.html] documentation. Some sample values for the xml:lang=attribute are:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>en</td>
</tr>
<tr>
<td>French</td>
<td>fr</td>
</tr>
<tr>
<td>German</td>
<td>de</td>
</tr>
<tr>
<td>Italian</td>
<td>it</td>
</tr>
<tr>
<td>Latin</td>
<td>la</td>
</tr>
<tr>
<td>Language</td>
<td>Code</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Arabic as spoken in Iraq</td>
<td>ar-IQ</td>
</tr>
<tr>
<td>Chinese</td>
<td>zh</td>
</tr>
<tr>
<td>simplified Chinese</td>
<td>zh-Hans</td>
</tr>
<tr>
<td>Taiwanese</td>
<td>zh-TW</td>
</tr>
</tbody>
</table>

If further explanation is required, a `<language>` element with an `ident` attribute of the same BCP 47 code can be specified in the TEI header.

Copyleft 2008 Syd Bauman and Julia Flanders; source available at [http://www.wwp.northeastern.edu/outreach/seminars/_current/handouts/elementList.tei](http://www.wwp.northeastern.edu/outreach/seminars/_current/handouts/elementList.tei)
# Oxygen 'Shortcuts'

(With thanks to the Women Writer's Project, Brown University)

<table>
<thead>
<tr>
<th>in order to ...</th>
<th>in oXygen ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert an element</td>
<td>type <code>&lt;</code>, and then scroll through the pop-up list of available element names; note that typing the first few letters of an element's name will select it</td>
</tr>
<tr>
<td>insert an attribute</td>
<td>position your cursor before the <code>&gt;</code> of a start-tag (or before the <code>/</code> of an empty-element tag) and type a space, then scroll through the pop-up list of available attribute names; note that typing the first few letters of an attribute's name will select it</td>
</tr>
<tr>
<td>validate</td>
<td>press <code>cmd-shift-v</code> (or <code>ctl-shift-v</code> on PCs)</td>
</tr>
<tr>
<td>indent (i.e., make pretty)</td>
<td>press <code>cmd-shift-p</code> (or <code>ctl-shift-p</code> on PCs)</td>
</tr>
<tr>
<td>make text into an element (i.e., surround it with tags)</td>
<td>select text of interest, and press <code>cmd-e</code> (or <code>ctl-e</code> on PCs), then scroll through the surround window's list of available element names; note that typing the first few letters of an element's name will select it</td>
</tr>
<tr>
<td>rename an element</td>
<td>place cursor on either start- or end-tag, and press <code>opt-cmd-r</code> (or <code>alt-shift-r</code> on PCs)</td>
</tr>
<tr>
<td>split element</td>
<td>press <code>ctl-opt-d</code></td>
</tr>
</tbody>
</table>
What is the TEI?

The following can be found at www.wwp.northeastern.edu/outreach/seminars/readings.html

The Text Encoding Initiative is an attempt to prevent the digital equivalent of the Tower of Babel. It proposes a community standard for representing digital texts in a way that is both powerful and responsive to the research needs of humanities scholars.

In practical terms, it is several things:

• it is a set of guidelines for encoding humanities texts using XML, the eXtensible Markup Language
• it is a standards organization that maintains and develops an XML encoding language for encoding humanities texts;
• it is an international consortium that exists to support this standards development work
• it is a community of projects and individuals who use the TEI Guidelines

Like many standardization efforts, the TEI faces the challenges inherent in such a project.

How can the many disciplines and communities within the humanities domain find common ground in a single encoding language? How do we agree on the level of detail that is necessary or appropriate in describing our textual materials? How do we reconcile the advantages to be gained by consistency and agreement with the need for individual specialization? How do we handle the truly idiosyncratic and unexpected?

Unlike many standardization efforts, the TEI addresses these and similar questions by explicitly accommodating variation and debate within its technical framework. The TEI Guidelines are designed to be both modular and customizable, so that specific projects can choose the relevant portions of the TEI and ignore the rest, and can also if necessary create extensions of the TEI language to describe facets of the text which the TEI does not yet address. Because the TEI itself is complex, the customization process is not entirely trivial, but it is designed to be as straightforward as possible.

What is it used for?

For sheer quantity, the predominant use of the TEI Guidelines is in creating large digital library collections, such as those being developed at the University of Michigan's Humanities Text Initiative (for example, the American Verse Project), or the Electronic Text Center at the University of Virginia. Collections of this type provide access to large quantities of textual material, often focusing on rare or fragile materials which would otherwise be inaccessible. The text encoding in collections like these emphasizes features that will be of immediate, general use for searching and retrieval: information such as bibliographic data, basic text structure, and metadata such as...
subject keywords to help users find materials of interest.

The TEI Guidelines are also used by more specialized research projects to represent smaller, more thematically focused collections of texts, often organized around a single genre, author, period, or country (or a combination of these: for instance, the Italian Women Writers Project at the University of Chicago). Projects of this type often use more detailed markup to represent particular features of the text that are relevant to the specific collection or important for the specific scholarly audience being served. For instance, a collection focusing on an author whose writings are an important source of information about famous contemporaries might usefully encode all references to names and works, perhaps including links to more detailed indexes of biographical or critical information. Similarly, an electronic edition of a particular author or work might well include a representation of variant readings, authorial revisions, editorial emendations, and similar editorial information. Some collections of this sort are intended to serve very specific research goals, such as linguistic analysis, or to serve as the basis for a dictionary, and in these cases the markup may be very highly specialized.

The uses described above are all some species of publication: the goal is to create a digital collection that can be used online by some public audience of greater or lesser extent—perhaps limited to a small community or to paying subscribers, perhaps open to the general public. More rarely, the TEI is also used by individuals to create digital representations of textual materials to support their own personal research, in forms which may or may not be published.

Where the scope and purpose of the larger collections may be determined by audience and funding, in the case of an individual's work the constraints are more personal and professional: the encoded material might serve as a private research tool, or might develop into the equivalent of a digital monograph that represents the author's analysis of a set of texts. The use of the TEI Guidelines in these cases may be as detailed as the author finds useful—limited only by time, energy, imagination, and the constraints of usefulness.

How do you learn about the TEI?

The TEI web site is a good source of general information about the TEI, and is also the best place to find the TEI Guidelines themselves, as noted above. However, the TEI web site is only one of many sources of useful information about how to use the TEI and how to understand its significance.

For those who are interested in the TEI at a general level—those who need to understand what it is, what it does, and why it’s important—there are a few different ways to begin. Although there is no single source that can give a complete picture, there is now a growing literature on the role of text encoding in scholarly work. The bibliography that accompanies this site attempts to list some of the sources we have found most useful. The WWP’s Guide to Scholarly Text Encoding also includes a general overview of the central issues. Perhaps the best way to get an initial grounding in the TEI is to attend a workshop. The workshops in the WWP’s 2007—2008 series, funded by the NEH, are specifically aimed at providing this kind of conceptual overview, coupled in some cases with
hands-on experimentation, but aiming above all at examining how text encoding in general, and the TEI in particular, connect with basic scholarly concerns and practices. However, even a hands-on workshop that focuses on technical specifics can be a very good way to get a basic understanding of what is at stake. While the specific details of the encoding language may not be important for your work, they can give a fuller sense of how text encoding operates as a representational strategy, and of what it can and cannot do.

Hands-on workshops are offered each year in the following programs:

- University of Victoria, Digital Humanities Summer Institute
- University of Virginia, Rare Book School
- International Congress on Medieval Studies at Kalamazoo (two introductory TEI sessions in 2007)

TEI workshops of various sorts are also held fairly regularly at Oxford University, Brown University, the University of Illinois at Urbana-Champaign, the University of Maryland, and other places as well. (If you know of additional events that should be listed here, please send email to WWP at brown dot edu.)

For those who need a more detailed understanding of the actual encoding process, the above workshops are a good start, but they are not sufficient on their own. Learning the TEI is like learning a language—it has a fairly extensive vocabulary and a complex range of usage. An introductory workshop will give you a good understanding of what the language is for and of its basic terms, but you need to follow it with both practice and a detailed exploration of how the language is really used. Having a project you can work on—a set of documents that interest you and will motivate your researches—is a great help. To gain a detailed knowledge of the TEI Guidelines as you encode, there are several things you can do:

- Read the TEI Guidelines, starting with the basic chapters (chapters 6 through 10), and extending to the chapters on more specialized topics. The most immediately relevant for most encoding work are chapter 14 (which covers linking and the alignment of parallel texts), 15 (which covers methods of representing analytical and interpretive perspectives on the text), 17 (which covers the representation of uncertainty and statements of responsibility), 18 (which covers the transcription of primary sources and of manuscripts in particular), 19 (which covers the encoding of critical apparatus), and 20 (which covers the detailed encoding of names, places, and dates).
- Read and post questions to the TEI listserv, and search the TEI-L archives for discussions of topics that interest you. TEI-L contains a very detailed record of a huge range of encoding issues, some abstruse and some very basic; the range of opinions and approaches can give you a valuable sense of how different kinds of projects use the TEI.
- Look at the work of actual text encoding projects. Many projects have documentation and some have exceptionally good documentation that explains
the rationale behind their encoding decisions and the criteria by which they recognize and encode specific textual features. We list some exemplary documentation on our readings and resources page. Ask questions; most encoding projects are happy to help those who are new to the field and glad to find that their experience is valuable. Many are also happy to share sample encoded texts, which can be a very useful way of getting a more detailed view of the encoding landscape.

What kind of TEI knowledge is useful, and for whom?

The modern academy is full of digital research tools and online collections, and humanities scholars and students increasingly depend on digital materials—primary sources, research databases, online journals, and the like. Among these collections, the ones which take the editing and scholarly integrity of the digital text most seriously are increasingly using TEI. Large-scale initiatives such as Early English Books Online Text Creation Partnership are using the TEI as their underlying encoding model, and the major digital editions now being produced typically use TEI because it offers the most nuanced and rigorous approach to describing the complexities of the edited text currently available. Some understanding of text encoding in general, and of the TEI in particular, is thus becoming the equivalent of an understanding of standard editorial practices a few decades ago: helpful for anyone who works closely with texts, and essential for anyone whose research depends on a critical awareness of how scholarly sources are produced and consumed.

Although comparatively few scholars currently are actively engaged in creating digital materials of this type, that may change in the near future. Digital publication is becoming more common, even though there are still strong institutional obstacles (such as the still scanty formal recognition of digital publication for tenure and review). As these obstacles diminish, and as the tools for digital publication become more widely available and easier to use, scholars may find that involvement in digital publications of one sort or another—digital editions and anthologies, digital monographs, collaborative digital research projects—is both feasible and attractive. As a consequence of this shift, they may also find that the scholarly discourse is increasingly conducted in digital modes. To whatever extent this becomes true—perhaps slowly and only partially—those who want to retain a full engagement with their colleagues' work and with the research materials available will benefit from an understanding of the underlying concepts and technologies involved.

It should be emphasized, though, that this understanding need not be at a technical level. The most significant concepts of text encoding, from a scholarly standpoint, are not the technical details but rather the broader ideas about modelling textual information, representational strategies, and editorial method: in fact, the same domain that has been the province of scholarly editing for centuries. What needs to be grasped is how these ideas translate into the digital medium, and what changes when they do.

For scholars who are planning a more active involvement in a TEI project—possibly as an advisor or director—these broad concepts are crucial but in addition it will be important to
understand how the TEI specifically represents textual information: what kinds of features it handles most naturally, what features require additional work or customization, what kinds of customizations are appropriate and how the customization process works. These issues are important because they have an impact on the guidance of the project and on basic practical matters such as what kinds of staff are needed and how much work they will need to do.

Some scholars may also be interested in experimenting with TEI encoding on their own, perhaps to begin an edition of a particular text, or to create a digital text for their own research purposes. The easiest way to get started in such a case is probably to take a workshop, but you can also use the materials that are available here to get oriented.

Clearly those who have technical responsibility for designing and developing TEI publications need to know far more. In addition to an overall familiarity with the TEI language and its range of usage, they need a strong understanding of how the TEI customization works and of the tools that are used to manipulate TEI documents and schemas. Information on these topics is available from the TEI, both at the TEI web site and at the SourceForge site where the TEI maintains the TEI Guidelines. The TEI listserv (TEIa-L@listserv.brown.edu) is also a good place to consult with experts who are willing to help.
1 Getting started using TEI: What is TEI?

Abstract: This article is the first article in a series that should help its readers getting started in the TEI. It explains the principles of text encoding and explains what to expect from the promised series of articles. It examines the reasons for using or not using TEI, and gives a full, albeit simplified, example of a real TEI text, showing the overall structure of a TEI document: header, facsimile, text consisting of front, body and back, and textual divisions. It also shows a number of optional extras that can help explain the text (notes), reconstructing the text's history (addition and deletion), or indexing the text (names and person description).

The article you are reading is the first installment in a series of articles that should help you getting started using TEI. The following instalments will appear in upcoming issues of TEI-EJ. The series cannot provide a full tutorial, and it cannot explain all of the detail of the underlying technologies, but it should make life a little easier for someone who is confronted with the TEI (Text Encoding Initiative) for the first time. This article begins by giving a brief explanation of the TEI (1.1 TEI: a very high-level overview), and goes on to describe what you can expect from this article series (1.2 What this article series does, and what it does not). We then discuss the question of when to use and when not to use TEI (1.3 When to use and when not to use TEI?). We give a first example of a complete TEI text in 1.4 Overall structure of a TEI text.

1.1 TEI: a very high-level overview

If you are reading this article you have probably decided, or are considering, using the TEI to create a digital version of one or more texts. Someone may have told you you should do so, or you may have read about other projects that used the TEI. So what is TEI, and why is it important?

A common answer is to that question is that the TEI is a set of Guidelines for the encoding of text. But TEI is not just a set of encodings and Guidelines for using them. The TEI is also an organization that maintains these Guidelines, and a community of users that applies the Guidelines. In fact the users are the organization: the TEI is a consortium of universities, libraries and research organizations, and the TEI Board and Council are elected by the member organizations. Using TEI is taking part in the community of TEI.
users that meet on the TEI mailing list, in Special Interest Groups, at the yearly TEI member meeting and on other occasions.

But what is Text Encoding? Text encoding is the addition of codes to text in order to make it possible for a computer to process that text. Text encoding assumes, usually, that you already have a text, either by manually transcribing it, or by using an OCR program. The codes that you add to the text will describe some aspect of the text: either by adding information such as author or place of publication (metadata), or by characterizing the role, meaning or other properties of pieces of text. The title of this chapter, to give a very simple example, might be encoded as:

```<head>TEI: A very high-level overview</head>```

You might ask why we do this. Word processing software, like Microsoft Word, can after all handle text very well without us having to type codes in pointing brackets. The long answer is given in another article in this series (The rationale of declarative markup), but a shorter answer is that explicit coding gives you (the editor of the text) control over your text and its application, something you don't have when you use a commercial word processor. For an office document that may not represent a problem, but for a text that you edit you probably do not want to depend on the standard facilities that were created to handle business documents. And while it is true that there are programs that can compute wordlists and complex concordances, without any need for explicit codes, that is only true for individual texts without structure and only provides limited means for interrogation of these texts. Once you have multiple texts you need a way to associate texts with titles, and if your text consists of poetry with a prose commentary, you may want to have word counts by type of text. That is to say: you need a structured representation of text properties, and that is what text encoding provides.

If you know something about web pages, you may think by now that text encoding is like using HTML, the language of the web. Isn’t the `<head>` that we saw just now very much like the various `<h1>`, `<h2>`, ... elements that HTML uses to encode headers at various levels? And indeed, the encodings that the TEI uses (so-called tags between angle brackets), is very similar to HTML's formalism, and some of the tags it shares with HTML (such as `<p>` for paragraph and `<div>` for textual division). The formalism is called XML (more about it in the Technical Background article in this series). What makes TEI different from HTML is that the TEI provides tags for many different text types (e.g. verse, drama, dictionaries, spoken texts) and many different sorts of properties (to document e.g. a bibliographical description, or textual genesis), and that each of these tags has an explicit, well-documented meaning. To give an example: you can represent a play in
HTML, but HTML has no special tags to indicate a speech, or a stage
direction. TEI does have these, and a computer program that reads a TEI
representation of a play can use that knowledge, e.g. to create an index of
speeches by character or search only within a certain character's lines. It
would be significantly more difficult to do so from an HTML representation.

Your next question might be: so, once I have a TEI-encoded version of my
text (i.e. an XML file that uses TEI tags), can I put that text on the web? The
answer is: yes you can, but not without one further step. You need some
sort of program that can turn your encoded text into a web page or
collection of web pages. Remember, during the encoding process you
explicitly describe the structure of your text, such as verse lines, headings,
notes, etc. But you say nothing about how you want these to be represented
on the web, or how you want them to be printed. There is a very good
reason for this. For most texts, there is not a single best way to represent
their content. One of the reasons why we do text encoding is that we want
to do multiple things with the texts (e.g. show them on the web, present a
search facility that can search by chapter, index personal names, etc.); for
that very reason, presentation functionality is kept outside of the text. The
idea is that the encoding should not be created for a single application
program. Programs come and go; the encoding remains.

There are many programming languages and other tools that can be used to
create a web page out of an XML file, but by far the most common is XSLT.
We apply what is called an XSLT stylesheet to an XML file to transform it into
one or more web pages. XSLT stylesheets can do much more than apply
what one would ordinarily call 'style' to a document. They can define
complex transformations including sorting, grouping, filtering, aggregation,
and selection. The TEI provides a set of general sample stylesheets that can
handle many types of text, but usually some stylesheet programming is
necessary to create a suitable set of web pages for a project. The amount of
programming depends very much on the ambition level and the properties of
the text. More about XSLT in the Technical Background and Running
stylesheets articles in this series. (By the way, there is also another type of
stylesheet, viz. CSS stylesheets. CSS stylesheets define simpler rendering
styles for XML or HTML (web) pages, including things like margins, colours,
font sizes, etc.)

Two more questions that often come up when discussing TEI are (1) must I
create this encoding by hand? And (2) can I modify the TEI encodings and
add my own tags? As to the first question, the answer is: to some extent.
Usually, people create XML files with a program called an XML editor (more
in the Choosing and installing an editor article in this series). Most modern
XML editors can help you create the encoding, based on a schema. The
schema defines what tags are allowed and where they are allowed, and the XML editor can prompt the user with a list of the allowed tags. The program can also check whether the tags that are used are actually allowed, in a process called validation. Some XML editors can show the definitions of these tags during editing, or adjust the layout and formatting of the text based on the tags that are used. Some offer a tagless interface for editing. All of these help very much in making XML editing a fairly straightforward process, depending on your background.

As to the second question: yes, you can modify the TEI encoding when your text has properties that the existing encodings really do not handle adequately. You can describe the extra tags that you need, or the ones that you need to change, in a special document called a TEI ODD file. The TEI provides a web application called Roma (http://www.tei-c.org/Roma/) which helps create such a document. Roma can also generate a schema from the ODD document. That schema, as we just saw, helps you create and validate your TEI document. Much more about this in the Defining, Creating and Using TEI Schemas article in this series.

The technologies that we have mentioned in this subsection (TEI, XML, HTML, XSLT, Schema’s, and XML editors) are more or less what will be discussed in the remainder of this series of articles.

1.2 What this article series does, and what it does not

If you have toyed around with downloading computer programs, you will have read 'Getting started with xyz' documents. Usually they are one or two pages long. The Getting Started-series that you are reading now may be about 50 pages long. That means that, depending on your purposes, it will probably take you longer to get started using TEI than it takes to get to use your average computer program. The reason for this is that you are not just learning to use a computer program; you are learning a new view of text, a vocabulary for expressing that view, a formalism for writing that vocabulary, and a set of programs for manipulating that formalism. The effort will depend on your existing skills, on your environment (do you have people around that can help you when you're stuck) and on your ambitions.

If you have had no previous experience of some of the more technical aspects of computing, getting to use TEI is going to present you with a sometimes steep learning curve. But then, having learned to use TEI will not just enable you to encode, present and analyse your documents, you will also have acquired a deeper understanding of your texts, and text in general, you will have mastered a set of technologies that will be of use.
throughout your professional life, and you will have met with a crowd of friendly and innovative scholars, programmers and librarians that may also prove to be an asset in your career as well as a reward in itself.

So, recognising that learning TEI is never going to be easy, this series of articles will help you get started. It is not a full course. Neither can it provide a substantive tutorial in the main technologies that TEI uses, such as XML, XSLT and HTML. We refer to tutorials and online training opportunities elsewhere wherever possible.

The article series aims to serve a diverse audience. Its readers may include academics (postgraduate, PhD, researcher, professor) who want to start understanding and using the TEI: e.g. a graduate student who wants to make an edition of a book he/she is writing a thesis on. The audience may also include people who are not themselves scholars but who are, or will be, working as encoders with a project. These may be undergraduates, editorial assistants, etc. We assume readers that are willing to learn something new, ready to work hard and to try various solutions. We also assume someone that is reasonably comfortable using computers, but do not expect experience in programming or previous knowledge of XML or HTML.

By 'reasonably comfortable using computers' we mean, among other things, that we expect you to have basic computer skills, such as create disk folders, move files, download and install programs, and use zip/unzip software. There is no point in this document trying to explain very elementary computer tasks. If you feel your basic computer skills are inadequate, you may want to attend elementary computer training or study a beginner's tutorial.

After this initial article, the following article will describe a number of technologies that the TEI uses in text encoding (2 Technical background). The reasons for encoding properties of the text rather than the desired output are examined in 3 The rationale of declarative markup. The series then goes on to discuss (in 4 Choosing and installing an editor) XML editing software, the principal tool for creating and modifying TEI texts. Once we have an editor we can load, modify and validate TEI documents (5 Load, modify, validate a complete ready-made document). We use stylesheets (6 Running stylesheets ) to transform XML documents into other documents, e.g. to read them on the web. Once you have seen this, you will want to begin using this on your own texts (7 Getting this to work on sample of own text).

A final piece of technology that is explained in the series are XML schemas 8
Defining, Creating and Using TEI Schema's. By using schemas we describe exactly which elements can occur in a given XML document, and what elements they may contain. Using the description, we can check the validity of the XML document. That is one of the reasons that we want to define schemas that closely fit the documents that we are encoding. The TEI provides a mechanism for creating such schemas. What remains then is to ask what to do once you have studied the series (9 Where to go from here). The article series comes with a Glossary (getstartgloss.xml) of technical terms.

This article series should be useful irrespective of the computer platform you are using (Windows, Unix, Apple). In fact, an important reason why the TEI was created was that computer files tended to be specific to a single platform. Work that scholars did to encode historic texts was lost when platforms changed. Platform independence is therefore very important to the TEI. The software that we discuss runs on all major platforms, though things may not look exactly the same everywhere. Where we have to describe things that are specific to one or more platforms, we will clearly indicate this.

1.3 When to use and when not to use TEI?

The TEI Guidelines have been in development over a period of more than 20 years. They have grown into a quite formidable apparatus for tackling textual issues that may appear intimidating at first. It is not uncommon for people to wonder: do I have to do this? Isn't there an easier alternative, something closer to the techniques that I already know? This section will discuss situations when using TEI is appropriate and when it is less so.

To be clear, you can encode any form of text-bearing object, and even objects that don't have any text, in TEI. People also use the TEI to record only metadata about objects. If you are undertaking a text-encoding project in linguistics, social sciences, library sciences, or humanities then it is likely that the TEI is a good fit for your needs. Nonetheless there are situations where the use of TEI may be overkill or otherwise less appropriate.

1.3.1 Why not just HTML?

Quite frequently people know that their main publication output will be in HTML on the web, and they question whether they should not just author directly in HTML. As mentioned earlier in this document the TEI has more and more specific elements than HTML. Experience has taught that it is more useful to describe what something is, rather than what it looks like.
Therefore, while in HTML encoding you might want to display in italics something that is a title, emphasized, a foreign phrase, or a specialized term, in TEI one would mark this with specific elements for titles, emphasis, foreign phrases, or terms. The benefit of this is the easy ability to repurpose the material for different uses. Also, by separating out the presentation from the encoding it allows one to easily change the form of presentation for all similar elements at a later date. Another example would be line numbers. If you edit in HTML, you either use line numbers or you do not. If you use TEI you mark the line-breaks in your text as such, creating the basis for producing an HTML text with line numbers and one without.

1.3.2 What is your material like?

For many genres (prose, poetry, plays, dictionaries, charters, language corpora and more) examples exist of successful TEI encoding. The TEI website contains a list of projects using TEI that can inspire prospective encoders. But one reason not to use TEI could be that for your genre or discipline there is an existing standard which suits your needs better. For example, if you are encoding a mathematical manuscript then the Mathematical Markup Language (MathML; Carlisle 2003) recommendation from the W3C is mostly likely a better choice. That said, if you are editing documents which contain mathematical formulas but also exhibit textual variation, TEI would be a better choice, as you can encode the textual variation using TEI features and embed MathML in your document to describe the formulas. This ability to be able to incorporate other standards as part of a TEI document is one of its many strengths. Many other standards exist and the TEI has tried not to reinvent the wheel where a perfectly good one exists, instead allowing you to embed markup from other vocabularies in your TEI document where necessary. Additionally if the TEI does not cater for some textual phenomena important to your work, then it is possible and even recommended to extend the TEI (in an approved and documented manner) to add new elements to deal with these.

Another technology that people use frequently instead of the TEI, or other forms of XML, is relational databases. It makes sense to use a standard relational database when the nature of the data benefits from this form of storage and retrieval. One way to think about this division of information is whether it is data-like or document-like. Relational databases are good at storing single static fields where no markup is present in any of the fields. XML is good at deeply-nested, arbitrary, fields. An example might be an address and telephone list. If one is simply recording a flat list of names, phone numbers, and some address lines then a relational database is possibly certainly sufficient. If the individual fields contain anything you want
to record inside any of them then using a structured markup language like TEI XML is probably more suitable. So-called 'document-like' information is often characterised by multiple levels of nesting of structures, for example divisions containing paragraphs containing various phrase-level markup such as names, abbreviations, or foreign phrases. See Bradley & Short (2005) for a discussion of some of these issues.

In some cases, source material may contain structures which cannot be represented in TEI markup (say you are visually representing a ballet performance). In these cases TEI is clearly not the answer (though it might be an ingredient in a solution). However, it may also be the case that your material could be represented in TEI but the discipline you are working in has a significant number of tools and expertise available for another format. This will necessarily be a factor in the format you choose.

### 1.3.3 What is your desired output?

It is one of the basic ideas of descriptive markup that the presentation and output should be left as a separate step from the encoding of the information. Part of the reason for this is that one is able to leverage the markup available in a TEI document to produce all sorts of outputs. (e.g. not only a rendering of a text, but indices of certain aspects, rearrangements or different views, and any number of statistics) The TEI provides basic customisable XSLT stylesheets for transforming TEI texts to HTML and PDF, and additionally there are ways to import and export TEI to/from word processors. One of the benefits of using XML as a format is that it is fairly easy to transform XML into other formats required for later software. The need to create output in multiple forms or formats is a clear reason for choosing TEI. Conversely, if your output is to be accessed only in a single format, other than TEI, then choosing that other format may be more suitable. Still, we would argue for carefully considering the potential advantages of multiple output forms.

### 1.3.4 How hard is learning TEI, and where do I get help?

The competence of the person undertaking the encoding is necessarily a consideration in choosing technologies. However, it should be emphasized that a fairly good understanding of basic TEI XML can be obtained quite quickly in comparison to many of the other skills people learn in order to undertake similar scholarly activities. Spending a few days learning TEI XML is an investment of time that is well worthwhile. That said, such an undertaking does presuppose a certain amount of basic computer literacy. It is worth remembering that in some cases it may also be possible to out-
source the required TEI encoding of your documents. One good reason for using the TEI is that there is a large international community who are willing to help you as you encounter problems. Nonetheless, if the context in which you are undertaking the encoding (workplace, institution, etc.) provides no support for TEI or XML, and actively supports alternative solutions, then that should certainly be borne in mind. What would be more beneficial, of course, is educate your colleagues as to the benefits of the TEI, and perhaps host a TEI Workshop with invited speakers to help train your local technical support. There are many scholars with basic computer literacy who have been more than able to teach themselves the TEI. There are also materials from many TEI Workshops available online.

1.3.5 Do long-term benefits outweigh initial costs?

One of the benefits of using the TEI is that what you do is specifying the textual distinctions that are useful for your purposes. This is intellectually more satisfying than only encoding for one platform such as web distribution. And yet, if you have no experience with the TEI, and simply need to do a single web page very quickly, then deciding to learn the TEI, and investigating the various options for transformation to HTML, is probably not as quick. We would argue, of course, that learning TEI is never time spent poorly.

1.3.6 But can markup do justice to my text?

There is a school of thought that holds that XML markup cannot do justice to the phenomena that literary scholars study. The main reasons for these doubts are that

- (i) XML encodes hierarchical structures, while most of the phenomena that literary scholars are interested in cross hierarchical boundaries: Quotations run across paragraph boundaries, speeches in plays overlap with metrical units in poetry, and textual variation is also insensitive to structural boundaries;
- (ii) XML encoding is often perceived as inflexible and rigid, not suited to represent the creative process of scholarly discovery;
- (iii) the need to learn and consider the technicalities of XML encoding is sometimes thought to be at odds with the scholarly work which the encoding should after all facilitate.

See for a forceful statement of these views e.g. Smith et al. (2006).

Should these criticisms keep you from using TEI XML? It should not come as surprise that we do not think so. This tutorial is not the right place for an in-
depth discussion of these fundamental criticisms of XML as a format. It is true that overlap between textual phenomena can be a cause of complexity in text encoding, but there are a numerous of ways of dealing with them and for the majority of people it is usually not a problem. As to the perception of inflexibility: it is clearly unwise to begin TEI encoding of a text without previous thought about the phenomena worth researching and worth tagging (Lavagnino 2006). However, when preparing a (collection of) text(s) for publication, some measure of uniformity is necessary, whether one uses XML or not. And in fact the use of XML can be argued to increase flexibility, as it facilitates multiple presentations of the text. Finally, while it is undoubtedly true that heavily marked up text can become difficult to read, this is more of an interface problem than an XML limitation. The problem may be alleviated by a tagless editing interface in XML editors or by editing applications that hide the existence of XML altogether. Another solution may be the use of so-called stand-off markup, i.e. markup that is not embedded within the text that it comments on but that points to segments of that text.

1.3.7 TEI and the history of digital text

Summing up, there is no single answer to the question when to use TEI. The decision will depend on characteristics of the texts, the discipline, the persons and the project involved. A final consideration may be the unique place of the TEI in the history of the development of digital textual studies, digital humanities and the way we conceive of electronic text in the modern age. As the TEI was originally a pre-web endeavour, its decisions have helped to influence many of the people who have designed the systems, formats and processes which make up the web, including XML itself (see DeRose 1999 and Barnard et al. 1995).

If embarking on a text encoding project of any sort, it will do one no harm to examine the current TEI Guidelines. They have been developed and updated over the last couple of decades with input from countless scholars and experts from many fields. They have been revised again and again to introduce new concepts, expand older ones, and correct mistakes. This does not mean that there is no room for improvement, but simply that they have benefited from a long history of many eyes and experience with all sorts of texts. The Guidelines then provide a good codification of knowledge about the markup of various textual phenomena. It is quite likely that your problems and concerns are not unique and that others have encountered similar textual phenomena before you. After all, damage to the carving on a stone tablet is very similar, in a structural sense, to damage to a text in a medieval manuscript, or indeed a modern printed book rescued from a fire. For many text types, if you decide not to use TEI, you are likely to have to
reinvent several wheels.

1.4 Overall structure of a TEI text

To introduce the structure of a TEI document, we will begin at the level of a smaller text fragment, and then gradually add the larger document structure around it. Our example is based on the Diary of Robert Graves. For educational purposes, the encoding that we will show here is slightly different from the encoding applied in that project.

1.4.1 A text fragment

Suppose we were to create an edition of the diary of Robert Graves. We have a facsimile of a diary page, which looks like this:

An initial transcription of the text might look like this:

Oct 10 Monday Ghost, completing ch IX Dictionary with Alan. A lot of time goes to making charcoal for 'Marthe', Beryl’s now using this fugon for warming her attic. Went to Montauban with David – first visit for about 10 days – ordered small wood for Dorothy's cresset. Now almost always win at Cambeluk: we are playing a correspondence game with Harry. Nono broke Laura's particular coffee cup, [sketch of cup] and she her blue glass bottle given by Karl.

The first thing to remark here is that in a diary transcription we will want to identify the date that the entry belongs to. We may also want to say that this piece of text actually is a diary entry. The next thing we notice is that the entry has a heading ('Oct 10 Monday') and consists of a number of paragraphs. We introduce some XML elements and attributes to account for these facts.

<div type="diaryentry" n="1938-10-10">
  <head>Oct 10 Monday</head>
  <p>Ghost, completing ch IX</p>
  <p>Dictionary with Alan.</p>
  <p>A lot of time goes to making charcoal for 'Marthe', Beryl's now using this fugon for warming her attic.</p>
  <p>Went to Montauban with David – first visit for about 10 days – ordered small wood for Dorothy's cresset.</p>
  <p>Now almost always win at Cambeluk: we are playing a correspondence game with Harry.</p>
  <p>Nono broke Laura's particular coffee cup, [sketch of cup] and she her blue glass bottle given by Karl.</p>
</div>
A fuller explanation of XML will be given in the Technical Background article in this series. For now, we will limit the explanation to saying that that XML elements such as 'p' (paragraph) are delimited by what is called tags. An opening tag looks like '<p>', a closing tag looks like '</p>'. In running prose, we use the opening tag in discussing the element. Elements can have properties attached to them by specifying attributes, as in the <div> element in the example. When discussing an attribute, we will prefix its name with '@', as in 'type'.

The <div> element is what is used to describe textual divisions. We use here the n attribute to indicate the date, and the type attribute to state this piece of text is a diary entry. <head> is used to describe a heading, <p> describes a paragraph. A question might be why we don't simply indicate paragraphs by newline characters, the way many programs do (such as Notepad on Windows). One reason is that different operating systems use different characters to indicate the beginning of a new line. Another reason is that an accidental new line character in our source would cause a rendering application to begin a new paragraph. The best way to avoid ambiguity and to make our encoding portable between platforms is to explicitly indicate new paragraphs. We might, in fact, even have indicated the locations where Graves begins a new line. TEI provides the <lb> element for that purpose. This, as most other decisions about encoding, is a matter of editorial choice. Is the beginning of a new line important enough to be retained in an edition? The TEI does not prescribe editorial policies.

Before moving on to the larger document structure, we will add a few more refinements. Not every project will find these refinements necessary, but they give an indication of the kind of phenomena TEI can handle. To begin with, the word 'fugon' is a word taken from Spanish, and should probably be italicised in an edition. We use the <foreign> element to indicate this. To explain what it means, we add a <note>. Then, there is a small drawing of a cup, which our transcription renders as '[sketch of cup]'. From the transcription, no-one would know that these are not Graves' own words. We will use a <figure> element and an embedded <figDesc> (figure description) to place the description. We will also indicate that 'ch' is an abbreviation (for 'chapter'). And finally, we will want to indicate a number of changes that Graves made in his text. We will use <del> and <add> elements to indicate deletions and additions to the text. The result is as follows:

```xml
<div type="diaryentry" n="1938-10-10">
  <head>Oct 10 <del>Tuesday.</del> <add>Monday</add>
</div>
```
Ghost, completing <abbr>ch</abbr> IX</p>
<p>Dictionary with Alan.</p>
<p>A lot of time goes to making charcoal for 'Marthe', Beryl's now using this</p>
<foreign>fugon</foreign>
<note>charcoal-burner</note>
for warming her attic.</p>
<p>Went to Montauban with David – first visit for about 10 days – <del>got</del>
<add>ordered</add> small wood for Dorothy's cresset.</p>
<p>Now almost always win at Cambeluk: we are playing a correspondence game with Harry.</p>
<p>No no broke Laura's particular coffee cup, <figure>
<figDesc>sketch of cup</figDesc>
</figure> and she her blue glass bottle given by Karl.</p>
</div>

Let's now put this fragment into context.

1.4.2 The larger document structure

Texts do not live in isolation. They are often introduced by title pages, prefaces, tables of contents and dedications; they are often followed by an index, appendices, and similar material. We represent this structure by a <text> element that contains a <front>, <body> and <back>. In our case, the editors of Graves' correspondence have created monthly abstracts of the diary entries. These might very well have been placed in the <front> element. Supposing for the moment that there is no other front matter and no back matter, that would give us this:

<text>
<front>
<div type="abstract">
<head>OCTOBER 1938</head>
<p>The rains set in, and Graves works in his bedroom with the fire going. ...</p>
</div>
</front>
<body>
<div type="diaryentry" n="1938-10-10">
</div>
</body>
</text>
Notice, by the way, the use of <!-- ... --> to write comments in XML text. Comments are used to explain something about the encoding to a human reader and will usually be ignored by programs that process the XML.

Apart from the texts that surround a main text, such as forewords and appendices, TEI texts are also provided with metadata, information about the text. All TEI texts come with a header that contains these metadata. The element is called <teiHeader>. The <teiHeader> and the <text> are children of the top level <TEI> element. So the overall structure looks like this:

```xml
<TEI xmlns="http://www.tei-c.org/ns/1.0">
<teiHeader/>
<text>
<front/>
<body/>
<back/>
</text>
</TEI>
```

Notice the 'xmlns' on the <TEI> element: all TEI elements are part of the TEI namespace, http://www.tei-c.org/ns/1.0.

We will not go into details here about the content of the TEI header, but show a very minimal example. Again, this is borrowed, with simplifications, from the Diary of Robert Graves.

```xml
<teiHeader>
<fileDesc>
<titleStmt>
<title>Diary of Robert Graves 1935–39 and ancillary material</title>
<author>Robert Graves</author>
<editor>...</editor>
</titleStmt>
<publicationStmt>
<publisher>...</publisher>
<pubPlace>...</pubPlace>
<availability status="unknown">p>...</p>
</availability>
<date>...</date>
</publicationStmt>
<sourceDesc>
```

Available: http://tei.oucs.ox.ac.uk/GettingStarted/html/in.html
The elements that are used here are largely self-explanatory. 
<availability>
is used to describe the (legal) conditions under which the text is available.
<sourceDesc>
describes the source from which the electronic document (the
TEI document) was created.

1.4.3 Two refinements

The encoding that we have shown up to now is of course only a very partial
encoding. We will show two ways of creating a more informative document.

1.4.3.1 Attaching a facsimile

First, we may want to make explicit the relation between the image file that
we transcribed and the transcribed content. For that purpose, the TEI
provides the <facsimile> element, stored between the <teiHeader> and the
<text> elements. The <facsimile> element describes the transcribed object
in terms of <surface>s and (optionally) <zone>s within these surfaces. In
our case, each surface will correspond with a diary page, and we have no
need for zones. For each surface, we can define one or more <graphic>
elements, which will hold information about the images of that page. The
surface element is provided with an xml:id attribute. From the transcription
we can point at the <surface> using the facs attribute. Let us see what this
looks like:

<TEI xmlns="http://www.tei-c.org/ns/1.0">
  <teiHeader> ... </teiHeader>
  <facsimile>
    <surface xml:id="graves1938-10-10-1">
      <graphic url="graves1938-10-10.jpg"/>
    </surface>
  </facsimile>
  <text>
    <front>...</front>
    <body>
      <div type="diaryentry" n="1938-10-10" facs="#graves1938-10-10-1"> ... </div>
    </body>
  </text>
</TEI>

You see we have added a <facsimile> element containing a <surface> for
the page that we transcribed. The <surface> has an xml:id attribute. On the
<div> element in the transcription we have added a facs attribute. The value of the facs attribute is a url. The value '#graves1938-10-10-1' points to the element with xml:id attribute 'graves1938-10-10-1', that is, the <surface> element. The <surface> element contains the <graphic> element that is an image of the current page.

The reason we do this is that we have now explicitly defined the relation between the transcription and the corresponding image files. Later users of the transcription will know what belongs together. What may be more important: applications that want to render the transcription will no longer need to know about the names of the image files. They can just fetch the files that are needed from the location as it is specified in the Guidelines. It is an important step towards making general-purpose TEI applications.

### 1.4.3.2 Repurposeable notes about persons

A further enhancement of the document's value would be to provide some explanations about the many persons mentioned, but not identified, in the diary entries. We could create <note> elements, the way we did to explain fugon, but most persons recur many times throughout the diary. What we would like to have is a single explanation per person, that we can show whenever it is needed.

One way to create such reusable explanations is to make use of the participant description in the TEI header. The <particDesc> element can contain a list of persons (a <listPerson> element containing <person>s). For each <person> we could, if we wanted to do so, give a structured description in terms of a number of characteristics (sex, age, etc). But we can also limit ourselves to an informal description using a <p> element. Whenever a specific person occurs in the diary, we can then refer to the description of that person in the header. The element that we will use to refer to the description is the <rs> element. The <rs> element (referring string) is somewhat like a name element, but more generic: names are referring strings, but so are 'her oldest son' or 'the gardener'. Which gives us something like this:

```xml
<TEI xmlns="http://www.tei-c.org/ns/1.0">
  <teiHeader>
    <fileDesc/>
    <profileDesc>
      <particDesc>
        <listPerson>
          <person xml:id="AH">
            <p>Alan Hodge. Oxford history graduate.
          </p>
        </person>
      </listPerson>
    </particDesc>
  </profileDesc>
</teiHeader>
```

Available: http://tei.oucs.ox.ac.uk/GettingStarted/html/in.html
Became close friends with Laura Riding & Robert Graves. First husband of Beryl Graves.


<p>David Reeves. Brother of James Reeves.</p>

For three persons in the diary fragment, we have created a <person> element. We have surrounded the names in the diary with <rs> tags and let the <rs> elements point to the <person>s using the ref attribute.
This concludes the discussion of our first TEI example. The choices made in the encoding of this diary fragment are not meant to suggest what is mandatory and what not: the editorial policies of individual projects must decide, e.g., whether to retain or to expand abbreviations or to do both. Similarly, a transcription that ignores deleted text can be perfectly valid TEI. Decisions about what to encode should be driven by research interest (and, inevitably, practical feasibility).

1.5 Summary

In this article, we have given an overview of the TEI for those that want to get started applying it. We gave a high-level overview of what TEI encoding is, and explained the thinking behind the series of articles of which this one is the first instalment. We discussed the circumstances under which applying the TEI makes sense, and when it does not. Finally, we gave an extended example of the overall structure of a TEI text. We saw that a TEI text always contains a header (<teiHeader>) that describes that text. The text itself (<text>) may, in addition to the text <body>, contain <front> and <back> matter. Additionally, a TEI encoded text may contain a <facsimile> element that relates the transcription to the pages that have been transcribed and to images of these pages. The transcription itself is usually structured using <div> element for the textual divisions, which in their turn will contain <p> elements (if we are dealing with prose).

We also saw a number of extras: we identified abbreviations, we encoded additions and deletions, and annotated references to persons. As said earlier, there is nothing in the TEI which obliges us to provide that information. Decisions about the whether to encode such features depend, among other things, on the text being edited, type of edition that is being planned, and the resources that are available. For those, however, that want to encode them, the TEI provides the necessary mechanisms.

But what exactly is XML, and why are we using it? Why do we provide our texts with this type of encoding at all? Why don't we use a wysiwyg editing environment, the way modern word processors do? How can we process files such as the one we saw? The next instalments in this series will deal in depth with some of the questions that we could only touch upon in this article.

1.6 Literature

Overviews of the TEI are given in Cummings (2007), Romary (2009) and Vanhoutte and Van den Branden (2010). A cogent case for using the TEI is

Available: http://tei.oucs.ox.ac.uk/GettingStarted/html/in.html
made by Sperberg-McQueen (2009). An argument against the use of embedded markup is made by Schmidt (2010).

4. Cummings, James (2007), The Text Encoding Initiative and the Study of Literature, in Ray Siemens and Susan Schreibman (eds.), Blackwell Companion to Digital Literary Studies (Blackwell: Malden (Ma)), 451-76.
2 Technical background

2.1 Text Encoding and XML

Strictly speaking, one doesn't need a markup language to share or even analyze digitized text. Computer programs can and have been written to take as input plain text and to analyze or transform it in various ways. Suppose, for example, you have the entire text of James Joyce's *Ulysses* stored in an ASCII computer file:

Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed. A yellow dressinggown, ungirdled, was sustained gently behind him on the mild morning air. He held the bowl aloft and intoned:

—Introibo ad altare Dei.

[etc.]

You can do a reasonably good job of breaking the text into its component paragraphs, sentences, and words based on line breaks, punctuation, and spacing. So you can produce word counts and concordances, calculate the average length of sentences, find interesting patterns of word collocation, or even generate a Joyce pastiche by stringing together random sentences from the text. But with only the bare text itself as data, there are many more things you can't easily do: search for references to people, places, or titles; distinguish reliably between primary text and quoted text, or direct and indirect speech; search for passages in a particular language, such as Latin; find text that is in verse rather than prose; indicate that text is in italics or boldface. For these tasks it is necessary to add something to the bare text, to mark it up: hence the need for a 'markup language'. Using a markup language you can identify "Buck Mulligan" as a personal name, and perhaps associate it with a standard identifier. You can indicate that the second paragraph represents a quotation, spoken by Buck Mulligan, in the Latin language, quoted from the Tridentine Mass, and typographically rendered in italics in the source text.

Various markup languages have been developed in the past to accomplish this kind of identification, but the Text Encoding Initiative and the wider computing world have settled on a single standard: XML, the Extensible Markup Language.
2.1.1 What Is XML?

XML is popularly known as an ‘angle-bracket language’. If you have ever looked at or edited HTML source code, you have worked with a near cousin of XML—or with pure XML, if your code was in the HTML version known as XHTML (Extensible HyperText Markup Language).

The most basic fact about XML is that XML is not a single markup standard, but rather a specification for creating markup languages, each with its own vocabulary and rules. Moreover, XML can be used to encode nearly any sort of material, from poems

```xml
<NurseryRhyme ID="mary_lamb">
  <title>Mary Had a Little Lamb</title>
  <stanza>
    <verse>Mary had a little lamb,</verse>
    <verse>its fleece was white as snow;</verse>
    <verse>and everywhere that Mary went</verse>
    <verse>the lamb was sure to go.</verse>
  </stanza>
</NurseryRhyme>
```

to regular ‘data’ of the sort that could be stored in a formal database:

```xml
<place>
  <name>London</name>
  <latitude hemisphere="N">51.507778</latitude>
  <longitude hemisphere="W">0.128056</longitude>
</place>
```

Nearly anything can be encoded and expressed in XML, within the constraints of its syntax. There are XML languages for musical notation [ref http://xml.coverpages.org/xmlMusic.html], for mathematical equations (MathML), for representing vector graphics (SVG), for library catalog records (MARCXML) . . . [ref to Wikipedia http://en.wikipedia.org/wiki/List_of_XML_markup_languages?]

2.1.2 XML Syntax in a Nutshell

The basic grammar of XML is simple enough that it can be expressed in four brief rules. (They are oversimplifications, strictly, but the exceptions are not important enough to matter at this stage.)

1. An XML file must have a single outermost root element that contains everything else. `<document> . . . contents . . . </document>`
2. An XML element must always have a start tag and an end tag. Both start and end tags are denoted by angle brackets preceding the name of the element; the end tag must have a solidus (/) preceding the
The two main things that an element can contain are text and other elements. 

4. `<para>This is a paragraph with text only.</para>`

5. `<para>This is `<emph>another</emph>` paragraph with a child element.</para>`

6. `</document>`

7. An XML element may qualified by *attributes*, contained within the start tag. The value of each attribute is contained within quotation marks (single or double): `<document type="legal" xml:id="DOC-2008-07-11-0003"> ... </document>` Within running text, attributes are referred to by prefixing an at-sign (@) to their name: type, xml:id.

8. XML elements must be nested; they cannot overlap. This syntax is illegal: `<line><clause>April is the cruelest month</clause>, <clause>breeding</line> <line>Lilacs out of the dead land</clause>, <clause>mixing</line> <line>Memory and desire</clause>, <clause>stirring</line> <line>Dull roots with spring rain</clause>. </line>` (The inability to capture "overlapping hierarchies" in this way is a fundamental limitation of XML.)

Any XML document that correctly follows these syntax rules is said to be 'well formed'. So long as XML is well formed, it can be parsed, edited, transformed, or otherwise processed by software tools. (Conversely, an ill-formed XML document will usually generate one or more error messages when opened by such tools, prompting the user to correct its syntax.)

Well-formedness by itself is not usually enough to make XML particularly useful to humans, however. For example, the following XML document is well-formed:

```
<NurseryRhyme>
  <verse>
    <stanza>Mary had a little lamb,</stanza>
    its fleece was white as snow;
    <stanza>and everywhere that Mary went</stanza>
    <stanza>the lamb was sure to go.</stanza>
  </verse>
</NurseryRhyme>
```

This attempts to encode a nursery rhyme, but it is lacking a title and switches the `<stanza>` and `<verse>` elements—we want the former to contain the latter, not vice-versa. And it has a verse line that lacks a tag entirely.

### 2.1.3 Rules for Structuring an XML Vocabulary
Fortunately, there are several ways of specifying rules for XML that allow one to create a structured XML vocabulary—a markup language that uses a defined set of elements and attributes that must be arranged in specified ways. When an XML document is compared against such a rule set and meets all of its requirements, it is said to be valid XML (in addition to being well formed). Most software that processes XML can check for both well-formedness and validity.

There are two basic technologies for writing XML rule sets: Document Type Definitions (DTDs) and XML schemas. DTDs were the earliest (in fact they go back all the way to the 1970s, when they were used with GML, the Generalized Markup Language, which is more or less the grandparent of XML); in loose usage, people sometimes use the term DTD to mean any rule set associated with an XML language: 'Have you figured out what DTD to use for encoding your recipes?'. XML schemas were developed in conjunction with XML itself, in the 1990s. Schemas are more powerful and more complex; their use in connection with TEI will be discussed below in [REFERENCE]. Here we will provide a simple example of how a DTD could be used to define the rules of NRML, our Nursery Rhyme Markup Language.

The rules for NRML are simple: an XML document encodes a single nursery rhyme in a root element called <NurseryRhyme>, which must have an identifier in an ID attribute. Under the root element there must be a title, contained in a <title> element, and one or more <stanza> elements; each <stanza> contains one or more verses in a <verse> element. A <verse> may contain only text.

Here is the DTD which specifies those rules:

```
<!ELEMENT NurseryRhyme (title, stanza+)> <!ATTLIST NurseryRhyme ID CDATA #REQUIRED > <!ELEMENT title (#PCDATA)> <!ELEMENT stanza (verse+)> <!ELEMENT verse (#PCDATA)>
```

With the explanation that "CDATA" means "character data" and "#PCDATA" means "parsed character data" (text that may contain special constructs handled by the XML parser), the syntax of this DTD should be fairly intuitive. The content model of an element is given in parentheses. Required elements are separated by a comma and must appear in the order given. A plus sign means "one or more".

Once a DTD or schema has been written for a particular XML markup vocabulary, it can be applied to an XML file by an XML validator to determine whether the file is valid or not. (The way this is done for TEI files will be discussed below [REF].)
2.1.4 XML Semantics

If you have followed the discussion of XML well-formedness and validity, it may occur to you that nothing in the rules for NRML that we have described prevents someone from tagging "Mary Had a Little Lamb" like this:

```
<stanza>
  <verse>Mary had a little</verse>
  <verse>lamb, its</verse>
  <verse>fleece was white as</verse>
  <verse>snow;</verse>
</stanza>
```

This markup is both well-formed and valid; no automated parser will ever complain about it. But a human reader is justified in calling it "bad markup", because the encoder has not correctly identified the boundaries of metrical verses. From a human point of view, the most important rule governing the `<verse>` element is that it should encode a single line of verse. The semantics of `<verse>` cannot be expressed in XML or an XML technology; they depend on the conventions of poetics. If one is writing a guide to the NRML language, it is not enough to assume the basic syntax of XML and to present the rule set (DTD or schema) governing its structure: the appropriate use of the `<verse>` tag must be explained in terms of the rules for recognizing poetic verse.

The same thing is true on a much larger scale with the TEI Guidelines. The TEI vocabulary extends to hundreds of elements, and the rules governing which elements may appear where are quite complex. But the largest part of the Guidelines consists of explanation, illustrated by examples, of which TEI elements should be used to encode various features of texts and their associated metadata. Often one needs to be an expert in a particular subject—bibliography, linguistics, manuscript editing—in order to use TEI encoding in a way that is meaningful as well as technically valid.

2.1.5 Namespaces: Avoiding XML Vocabulary Collisions

As the use of XML expanded and many different XML markup languages emerged, people realized that it would often be useful to permit one XML vocabulary to incorporate another one. For example, there is a formal XML language, MathML, that can be used to encode mathematical equations. If one were using TEI to encode the correspondence of a mathematician, rather than trying to extend the TEI tagset to include equations it would be
much simpler to use MathML directly within one's TEI document.

The first problem that arises when attempting to integrate differing XML languages is differentiating among the vocabularies, and in particular, avoiding name collisions. For example, suppose that we are encoding in TEI XML a scholarly work that has footnotes, which are tagged using <note>. And suppose that one footnote contains the opening bars of the song "Twinkle Twinkle Little Star". Let's say there is a widely used simple XML language called TuneML for encoding musical passages that also uses a <note> element, and we want to use it in our footnote. We might end up with something like this:

```xml
<note type="footnote" n="1">In the margin, the author has inscribed the opening bars of <title>Twinkle, Twinkle, Little Star</title>:</note>

```xml
<note>C</note>
<note>C</note>
<note>G</note>
<note>G</note>
</note>

[etc.].
</note>

We have an obvious problem. It's impossible to distinguish between the elements from TEI and from TuneML, and the <note> element is therefore used for two entirely different purposes.

The solution adopted to resolve this problem was XML Namespaces, a way of identifying the elements of a single XML vocabulary so that they occupy their own unique 'name space'. Namespaces are optional in XML; an XML document does not have to use them. None of the preceding examples of XML do, so they would be said to be in "no namespace". Namespaces give great flexibility and power to XML, but unfortunately they are one of the more confusing parts of XML for beginners.

A namespace is distinguished within an XML document using a unique identifier. And here is where the first confusion arises: namespace values share the same legal syntax as the Uniform Resource Identifiers (URI) used in the World Wide Web, and by convention they begin with http://. But a namespace identifier is not the name of a Web page, for all that it looks like one. For example, the namespace identifier of the TEI language is http://www.tei-c.org/ns/1.0. If you go to the Web address http://www.tei-c.org/ns/1.0, you will in fact find a short paragraph about the TEI namespace, but that is an optional courtesy on the part of the TEI. On the other hand, one of the namespaces used in Microsoft's XML format for its Office software is http://schemas.microsoft.com/office/word/2006/wordml, but there is nothing at the Web page.

A namespace must be declared in an XML document to take effect. It applies to the node it is declared in, and to all nodes below that unless they are declared to be in a different namespace. Thus, every TEI document begins

```xml
<TEI xmlns="http://www.tei-c.org/ns/1.0">
... content of TEI file
</TEI>
```

where xmlns= can be read as ‘the XML Namespace is...’.

Let's return to our hypothetical footnote above about *Twinkle, Twinkle Little Star*. Suppose we know that the namespace of the TuneML language is


We could then rewrite our XML footnote like so:

```xml
<note type="footnote" n="1">In the margin, the author has inscribed the opening bars of <title>Twinkle, Twinkle, Little Star</title>:
<tune xmlns="http://tuneml.org/schema/2009" clef="G">
  <note>C</note>
  <note>C</note>
  <note>G</note>
  <note>G</note>
</tune>
[etc.].</note>
```

Now we genuine have two different `<note>` elements. The first one is declared to be in the TEI namespace. The others, describing musical notes, inherit the TuneML namespace that is declared on their parent element `<tune>`.

### 2.1.5.1 Namespace Prefixes

If you have followed the preceding description of namespaces and their identifiers, it may occur to you that they are useful to computers but not very friendly to human readers and editors of XML documents. If you are looking at XML nodes deep within a file using multiple namespaces, how can you easily figure out which namespace is in control of the current node? And isn't it cumbersome to attach long namespace identifiers every time you refer to a namespace in a file?

The solution adopted in the XML world is *namespace prefixes*. When declaring a namespace, you may optionally associate it with a prefix. Then for the current XML node and any nodes below it, adding the prefix to an element name is the same as declaring its full identifier. The simplest strategy is to declare all namespaces with prefixes at the top of the document (i.e., on the root element):
Compare this example with the preceding one. Note that the TEI namespace is declared on the root element <TEI>, so it does not have to be repeated in descendant nodes; the footnote <note> inherits the TEI namespace. Also in the root element, the TuneML namespace is declared and associated with the prefix tn. Thus, within the file when elements from the TuneML language are used, they can simply be named with their prefix: <tn:tune>, <tn:note>. When using prefixes in this way, you must prefix every element you use, as there is no namespace inheritance—that is why every TuneML note is written as <tn:note>. But most people find XML documents with multiple namespaces more comprehensible when prefixes are used.

### 2.1.6 Further Reading

to come

### 2.2 XML on the Web

The ‘angle-bracket language’ that people are most familiar with is HTML, the Hypertext Markup Language that has powered the World Wide Web (WWW) since 1991. Its earliest format (still used today) was based not on XML but on the ancestor of XML, the Standard Generalized Markup Language or SGML. But since 2000 it has also existed as a pure XML language known as XHTML, which obeys all of the rules of syntax outlined above. For the purposes of this section, we are going to ignore the older format and use HTML to mean the XML form of the language.

#### 2.2.1 Web Servers, Web Clients

The WWW is based on a client-server model. That is, a Web server on the Internet hosts content and delivers it over the network to clients—to individual computers, intelligent phones, or other network appliances.
Oversimplifying some (but not much), every Web page view is the successful result of the delivery and interpretation of an HTML file with a structure like this:

```html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<xhtml:html>
 <xhtml:head>
  <xhtml:title>A Web Page</xhtml:title>
 </xhtml:head>
 <xhtml:body>
  <xhtml:h1>A Sample Web Page</xhtml:h1>
  <xhtml:p>This is a <xhtml:em>minimal</xhtml:em> HTML Web page.</xhtml:p>
 </xhtml:body>
</xhtml:html>
```

Chances are you're familiar with the syntax. Information about the document (metadata) goes inside the <head>, while the <body> contains the document text (plus references to graphics, multimedia, programs, etc., that display within the page). Heading levels are designated by <h1>, <h2>, etc., paragraphs by <p>, emphasized text by <em>, and so on.

The Web began its meteoric rise once people started writing client software that could interpret and display the HTML language in a standard way: the Web browser, or just ‘browser’ for short. From Mosaic through Netscape to MS Internet Explorer, Firefox, Safari, and all the other contemporary ones, a browser has had the function of interpreting HTML code (and bits of programming languages and other data embedded with it) and displaying it to a user in a standard way. So, by convention, the <title> of an HTML file is shown at the very top of the browser window; an <h1> head is usually in boldface type much larger than the regular text font; text inside an <i> tag is of course rendered in italics, and so on. The existence of a fairly limited set of uniform tags meant that Web designers could predict with some reliability how their HTML documents would appear at the recipients' end.

---

**Figure 1. Typical browser view of sample HTML**

### 2.2.2 Styling the Web

A piece of technology that has given the Web much of its presentational flexibility is Cascading Style Sheets, or CSS. As CSS is an indispensable tool for most projects using the TEI to present material on the Web, it is worth a brief detour here.
CSS is a *style sheet language* that can be used to precisely define the appearance of XML documents (not just HTML ones, note!) on screen, in print, or even for audio or Braille devices. It allows Web designers to go well beyond the default renderings given to HTML elements by Web browsers. Consider the sample HTML file given above. Typically, a browser will render the `<h1>` in boldface font at around 150–200% of the body font size, and will render the `<em>` as italics. But suppose I want my headings to be at 130%, in small capitals, colored green; and I want my emphasized text to be bold and red, not italicized. I can accomplish this by adding a `<style>` element to my HTML `<head>`, containing the relevant CSS instructions:

```html
<style type="text/css">
  h1 { font-size:130%; font-variant:small-caps; color:green; }
  em { font-weight:bold; color:red; font-variant:normal; }
</style>
```

*Figure 2. Typical browser view of sample HTML with CSS styling*

### 2.2.3 Limitations of HTML

In its origins, HTML was designed for the sharing of technical documents and other information generating by researchers. It has been expanded greatly since the beginning, but its legacy explains two key features of HTML: (1) a relatively impoverished semantic vocabulary for describing text structures, and (2) an emphasis on appearance over structure. Together with CSS, it has an extraordinarily rich array of mechanisms for sizing, coloring, and positioning text and graphics, but it has no native way of distinguishing between, for example, prose and poetry. Most of its structural or semantic tags describe features typical of technical documentation or memo-like prose (`<table>`, `<ol>` [ordered list], `<dl>` [definition list]), or exist to enable hyperlinking, form submission, etc. (`<a>`, `<form>`/`<input>`). It is a superb vehicle for *presenting* text, graphics, and multimedia, but an inadequate one for *representing* the underlying structure and meaning of the universe of human textual production.

### 2.2.4 So... Why not Just Put XML on the Web?

Let's return to our invented Nursery Rhyme Markup Language. Suppose I have used NMRL to mark up hundreds of nursery rhymes; can I just copy my files to the Web? Let's recall what an NRML file looks like; let's call it `MaryLamb.xml`:

```xml
<NurseryRhyme ID="mary_lamb">
  <title>Mary Had a Little Lamb</title>
  <stanza>
  </stanza>
</NurseryRhyme>
```

Mary had a little lamb,
its fleece was white as snow;
and everywhere that Mary went
the lamb was sure to go.

Figure 3. MaryLamb.xml viewed in Firefox
This is a ‘raw XML’ view of the file. Web browsers know how to display HTML because it has a known set of tags and styling commands. But without some hints, they cannot make any assumptions about how an unknown XML language should be displayed.

2.2.5 CSS to the Rescue?

CSS can define the appearance of any XML elements, not just ones from the HTML vocabulary. So it is entirely possible to take a newly invented XML language like our NRML and use CSS to display it on the Web. We do this by putting our CSS instructions into a separate file, and then referencing the CSS file from our XML file using an XML processing instruction. Let’s say we create a file NRML.css with the following contents:

```css
NurseryRhyme { display:block; margin:1cm; font-size: 14pt; font-family: "Bookman Old Style";} title { display:block; font-size:larger; font-weight:bold; margin-bottom:1em;} stanza { display:block; margin-bottom:1em;} verse { display:block; line-height: 1.3;} name { display:inline; font-variant:small-caps; }
```
The CSS display instruction tells the browser whether each element is to be displayed as a block (separated from surrounding elements, like a paragraph) or inline (part of the surrounding flow of text). We have defined margins plus a font size and style for the whole <NurseryRhyme> element, and added various spacing and typographic styling to the other elements.

With our CSS file done, we add the following code to the top of MaryLamb.xml:

```xml
<?xml-stylesheet href="NRML.css" type="text/css"?>
```
The result, viewed in a modern Web browser, looks like this:
So is this all we need to do to publish XML documents, whether in TEI or any other XML language, on the Web? The answer is again 'yes and no'. Yes, this is an effective simple way of displaying formatted XML; but no, it is not a flexible enough solution to handle several important needs when publishing on the Web:

- CSS instructions merely tell a Web browser how to display an XML file; CSS cannot add substantive content to the file. HTML files, for example, often include important metadata via `<meta>` tags in the `<head>`, which convey information about the file's character encoding, language, authorship, copyright status, and more. CSS cannot add `<meta>` tags.

- The version of CSS supported by most Web browsers can only tell browser how to display XML elements in the order they are encountered. It cannot reorder, transform, or apply special logic to the underlying data.

For example, suppose we want to publish our XML version of *Mary Had a Little Lamb* with the following enhancements:

1. each verse line is preceded by its line number in brackets
2. every second line is indented
3. the ID value of the document (`<NurseryRhyme>/ID`) is given in a footnote line following the verse
4. An HTML `<meta>` tag is added to provide the keyword "nursery rhyme" to be picked up by Web indexers like Google

All this can be done, but not by the CSS language. Instead, we must use a more powerful general-purpose programming language that can operate on XML data. Fortunately, there are well-established tools for this purpose, and they are usually indispensable for any project with a body of TEI-encoded texts that they want to share on the Web. The following section looks at one of the most commonly used transformation tools.

### 2.3 Transforming XML

The XML language was formally proposed by the World Wide Web Consortium (W3C) in 1998. As the language was being developed, it was recognized that for XML to be useful there would have to be programming tools available to query, transform, and render XML in various ways, whether for Web publishing or for other purposes. Concurrently, therefore, a working group developed specifications for an extensible stylesheet language: a kind of programming language that could be applied to XML documents to extract data, transform on type of XML file into another,
generate text output, or produce a print-ready document in an entirely
different typesetting language such as LaTeX or PostScript. The result,
formalized during 1999–2001, was XSL, the eXtensible Stylesheet Language.
XSL is the umbrella term for three specific language that do the actual work of transformation:

**XPath**
The XML Path language, or XPath, provides a way to identify and retrieve
specific parts of an XML document. For example, using our nursery rhyme
example, the `<verse>` element reading ‘its fleece was white as snow’ can be
extracted with the following XPath instruction: `//stanza[1]/verse[2]`. Or
we can return every verse containing the word *Mary* with the XPath
`//verse[contains(., "Mary")].`

**XSLT**
XSLT (which stands for ‘Extensible Stylesheet Language Transformations’,
though the full name is rarely used) is the workhorse of XML programming
languages. It is designed to take as input one or more XML documents, and
to enable a wide variety of operations on them in order to transform their
contents in virtually any way. XSLT is commonly used to transform content
from one XML language to another—for example, from TEI to HTML—or to
add or subtract features within a particular XML language. For example, we
could use XSLT to add internal line numbers to our NRML verse elements:

```xml
<verse n="2">its fleece was white as snow;</verse>
```

XSLT can also transform XML to plain text. XSLT contains many of the
structures of other general-purpose programming languages: variables, flow
control, if-then logic, and (via XPath, which it incorporates) many functions
that operate on strings and numeric values.

**XSL-FO**
XSL-FO stands for ‘XSL Formatting Objects’. It is a stylesheet language
designed specifically to apply document formatting to XML files primarily for
paged output, such as book publication. It is commonly used to convert an
XML file into PDF that can be displayed or printed.
The most common application of XSL in the TEI world is probably the use of
XSLT to transform TEI documents for display on the Web. In the remainder
of this section, we will give a realistic example of how XSLT can be used with
our nursery rhyme to achieve the exact Web publication format we want.

### 2.4 XSLT: A Practical Example

For reference, here is our nursery rhyme XML:

```xml
<NurseryRhyme ID="mary_lamb">
  <title>Mary Had a Little Lamb</title>
  <stanza>
```

<verse>Mary had a little lamb,</verse>
<verse>its fleece was white as snow;</verse>
<verse>and everywhere that Mary went</verse>
<verse>the lamb was sure to go.</verse>
</stanza>
</NurseryRhyme>

We want to publish it on the Web with the formatting we used in the CSS output example above, but we also want to (1) number verse lines, (2) indent every other line, (3) show the document ID in a footnote line, and (4) add a <meta> tag with the keyword "nursery rhyme". To accomplish this, we are going to write a short XSLT program that takes _MaryLamb.xml_ as input, and produces as output a single HTML file that contains all the content and style instructions needed. The result will look like this:

Figure 5. _MaryLamb.xml_ as transformed by XSLT, viewed in Firefox
The XSLT used to produce our new HTML file follows. Viewing XSLT for the first time can be intimidating, as it is a relatively verbose language. Here are a few preliminary comments and things to notice in the code:

- XSLT is itself written as an XML document. All of its basic instructions take the form of XML elements with the namespace prefix xsl. For example, where another programming language might create a variable called 'lineNo' with an assignment like _let $lineNo := number(line)_ in XSLT you would create the variable with an XML element: `<xsl:variable name="lineNo"/>
- `<xsl:number/>
- `<xsl:variable>`
- The root element of every XSLT program is an `<xsl:stylesheet>` element. It declares namespaces and sets other options. An `<xsl:output>` element can be used to specify output in XML, HTML, XHTML (as below), or text.
- The first workhorse of XSLT is the _template_, expressed in a `<xsl:template>` element. For transforming XML, templates are typically created for all or most distinct elements in the input document. They provide the rules for transforming each kind of element into something else. The real strength of XSLT is that it applies templates _recursively_, meaning that it can descend into your XML document and transform all its nested elements without your having to write any special program logic.
- The other workhorse of XSLT is the fact that you can create XML output simply by inserting an XML element in your program. If you look through the `<xsl:template>` sections below, you will see that in each case it is followed by a construct in HTML. For example, the first
template matches the NRML root element <NurseryRhyme>, and it
immediately constructs an HTML document template using <html>,
<head>, <body>, and so on.

Look over the script, and by comparing it with the input nursery rhyme and
your knowledge of HTML elements, try to get a general sense of what it is
doing. You'll find the template for <verse> is the most complicated. After
the script we'll present a bit of explication of what is going on in the
templates.

<?xml version="1.0" encoding="UTF-8"?> <xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
<!-- NRML-to-HTML.xsl: transform Nursery Rhyme Markup Language
into HTML --> <xsl:output method="xhtml"/>
<xsl:template
match="NurseryRhyme"> <html> <head> <title> <xsl:value-of
select="title"/> </title> <style type="text/css"> body { margin:1cm; font-size: 14pt; font-family: "Bookman Old Style"}
  h1 { font-size:larger; font-weight:bold; margin-bottom:1em;}
  div.id { font-size:80%; margin-top: 2em; line-height: 1.5em;}
  div.stanza { margin-bottom:1em; line-height: 1.3}
  p.verse { margin: 0; text-indent: 1.5em;}
  span.name { font-variant: small-caps; } </style> <meta
name="keywords" content="nursery rhyme"/> </head> <body>
<h1><xsl:apply-templates select="title"/></h1>
<xsl:apply-templates select="stanza"/>
</body> </html>
</xsl:template>
<xsl:template
match="stanza"> <div
class="stanza"> <xsl:apply-templates/> </div> </xsl:template>
<xsl:template
match="verse"> <xsl:variable name="lineNo">
<xsl:number level="any"/> </xsl:variable> <p>
<xsl:attribute
name="class"> <xsl:choose> <xsl:when test="$lineNo mod 2 eq
0">verseI</xsl:when> <xsl:otherwise>verse</xsl:otherwise>
</xsl:choose> </xsl:attribute>
<xsl:value-of select="concat('[',$lineNo, ''])"/>
<xsl:apply-templates/>
</p> </xsl:template>
<xsl:template
match="name"> <span
class="name"> <xsl:apply-templates/> </span> </xsl:template>
You should be able to see that for nearly every element in NRML, there is an
<xsl:template> section in our XSLT program. For example, when the
program encounters an NRML <name>, this is what it does:
<xsl:template
match="name"> 
  <span
class="name"> 
    <xsl:apply-templates/>
  </span> 
</xsl:template>
It outputs an HTML <span> element with class set to "name" (so that CSS
will format it appropriately). Then it applies templates using the <xsl:apply-
templates> element. <xsl:apply-templates> is the heart of XSLT recursion.
Essentially, it says this:

• examine the contents of the current element (here, <name>)
• if the element has children (child nodes) for which there are template rules elsewhere in the XSLT file, apply those rules
• if we encounter children for which there are no template rules, apply the default template: output the textual content.

Thus when our XSLT script encounters

```xml
<name>Mary</name>
```

it will output

```xml
<span class="name">Mary</span>
```

following the template rule for "name".

The template for <verse> provides an example of the power of XSLT. It sets a variable lineNo, which is simply the ordinal number of the current <verse> within the poem. It does this by calling on a built-in XSLT operation, the <xsl:number> element, which allows for highly flexible numbering. It then uses the $lineNo variable ('$' is the sign for 'variable') within a numeric function that tests whether the current verse is even or odd ($lineNo mod 2 eq 0) in order to assign the appropriate class attribute to the HTML <p> that controls whether or not the line is indented. $lineNo is also used to add the line number within [ ] preceding each verse.

XSLT is a powerful and complex language, and like any full-featured programming language it requires considerable learning time to reach productivity. But once learned, XSLT can be an extraordinarily productive tool for manipulating any text encoded as XML. For instance, it took the author of this section about half an hour to write the 50-line XSLT program above. Once written, it can transform any number of nursery rhymes encoded in NRML. Compare that against the labor it would take to convert, say, 1000 NRML files to HTML by hand!
Digital humanities

What is XML and why should humanists care? An even gentler introduction to XML

Introduction

Most introductions to XML seek to explain when one should use XML instead of HTML (generic, standard web pages), an approach that is most easily understood by those who already have some experience with HTML authoring. Other resources intended to introduce XML to novices, like the Text Encoding Initiative (TEI) Gentle introduction to XML, go beyond merely describing why humanities scholars might wish to use XML and what an XML document looks like, aiming also both to situate XML in a general context of markup languages and to introduce the syntax of the formal schema languages used to model them. Digital humanities scholars will need this information eventually, and perhaps even fairly early in their training, but including it in the very first introduction that complete novices will read entails the risk of inadvertently confusing or intimidating the new learner.

The present even gentler introduction aims to describe briefly some of the reasons humanities scholars use XML, and to do so in a way that will be accessible to absolute novice readers, who may have no idea what a web page looks like beneath the surface. It also describes the most basic features of XML documents, but it defers details about formal modeling and schema languages for another occasion. It ranges very broadly, and novice readers will find new terms (italicized when first introduced) and concepts in almost every paragraph, but I have tried to provide definitions, descriptions, and examples that will make the unfamiliar accessible even to those with no background at all in informatics. It is not necessary on a first reading to take notes or study this introduction; it is intended to begin to make new terms and concepts familiar, and some parts may be clearer than others on first acquaintance.

The title of this essay was stolen from C. M. Sperberg-McQueen (http://csmcq.com/1997/drh97.html), a pioneer and leader in the world of digital humanities, and the subtitle from the TEI Gentle introduction, where many digital humanists of my generation began their own learning.

The context

Digital humanities (and, in particular, the subfield of DH traditionally called humanities computing) is the use of computer technology to conduct primary humanities research. This introduction has been written for an undergraduate course entitled “Computational methods in the humanities,” a title chosen to emphasize the students’ creation of original programming for use in their own humanities research projects. This focus ignores other uses of computers in humanities scholarship, such as the creation of web pages, blogs, wikis, social networks, etc., not because those uses are unimportant, but because publishing something on the screen does not automatically move the researcher very far beyond what would be possible with paper publication, and because collaborative or social authoring on the Internet differs from other forms of collaboration more in ease of use than in the significance of the eventual research results. This course concentrates instead on three things that fundamentally distinguish digital humanities from non-digital humanities scholarship:

1. The creation of electronic texts that can be used in primary humanities research that would be impossible (or so impractical as to be essentially impossible) to conduct without computational assistance.
2. The use of existing computational tools to interrogate those texts and obtain original research results that would not be attainable without the tools.

3. Where existing computational tools are not able to meet the research goals, the development of original computer systems and programs to meet the researcher’s needs.

For reasons described below, most digital humanities projects nowadays (this document was first written in 2011) use XML as their document format. I describe below what an XML document looks like and why it has achieved the popularity it has as a way of modeling some of the inherent structural and semantic properties of the cultural documents used in humanities research. This is not the only use of XML (which is often employed to model other types of documents for other purposes, some of which have nothing to do with humanities scholarship), but it is the best place to start learning about why XML has come to play a uniquely important role in digital humanities scholarship.

Document and their structures

We often think of the documents we read as consisting of linguistic text that conveys meaning, but much of that meaning is expressed by the structure of the text, and not just by its linguistic content. For example, a book may be divided into chapters (coordinated with a table of contents in the front). The chapters, in turn, may be divided into paragraphs and may be preceded by chapter titles, and the text within the paragraphs may have footnotes, bibliographic references (coordinated with a bibliographic list in the back), etc. If a book were merely a stream of words, with none of the layout and formatting that lets us recognize its constituent parts, the reader would have much more difficulty determining where logical sequences of thoughts (instantiated as chapters or paragraphs, described by titles, etc.) begin and end. For that matter, although we don’t usually think of modern writing this way, the spaces between words are a fairly recent invention in human writing systems, they often do not correspond to pauses in spoken language, and they therefore serve as visual clues about the structure of the text as containing a stream of words, rather than just a stream of letters.

Chapters with titles and paragraphs are not the only type of organization found in documents. A poem might be divided not into chapters with paragraphs, but into cantos, stanzas, and lines. A play might be divided into acts, scenes, speeches (accompanied by the name of the speaker), and stage directions. A dictionary might be divided into entries, with headwords, pronunciations, definitions, examples (associated with specific definitions), etymologies, etc. If all of the words in any of these documents were simply run together in a continuous stream, the reader would have enormous, and perhaps even insurmountable, difficulty recognizing the structure—and the structure in these cases is part of the meaning. The name “Hamlet” means something different when spoken by a character in the play than it does when it serves as a label in an actor’s script, where it is intended not to be pronounced during a performance, but to identify the speaker of a particular line. A word in a dictionary has a different function when it is a headword, part of a definition, or part of an example, and the specific function of each word must be recognized if the dictionary is to serve its intended purpose as a reference resource.

The structures described in the preceding paragraph can all be considered ordered and hierarchical. That is, a single book may contain multiple chapters, in a particular order, where each chapter may contain one title followed by multiple paragraphs, also in a particular order. The chapter title does not normally fall between, say, the third and fourth paragraph, and paragraphs do not normally contain chapters or books.

XML is a formal model that is based on an ordered hierarchy, or, in technical informatic terms, a tree. It consists of a root (which contains everything else), the components under the root, which contain their own subcomponents, etc. These components and subcomponents are called nodes. In the case of our book example, the root node is the book, it might contain a title-page node, followed by a table-of-contents node (which contains nodes for each content item listed, in order, and each of those, in turn, might contain two nodes, one for the table-of-contents entry and the other for the page reference). The table-of-contents node might be followed by chapter nodes, in order, and each chapter node might contain a chapter-title node followed by paragraph nodes, once again in a particular order. Nodes may also contain text, and not just other nodes.

Humanities scholars represent their documents in XML for two reasons:
1. XML is a formal model designed to represent an ordered hierarchy, and to the extent that human documents are logically ordered and hierarchical, they can be formalized and represented easily as XML documents.

2. Computers can operate very quickly and efficiently on trees (ordered hierarchies), much more quickly and efficiently than they can on non-hierarchical text. This means that if we can model the documents we need to study as trees, we can manage and manipulate large amounts of data efficiently.

The beginning of every digital humanities project: Document analysis

The hierarchy we see in our document depends on our interests, and the same document might be amenable to more than one hierarchical analysis. A book in prose can be considered from an informational perspective as chapters containing paragraphs, but from a physical perspective it is pages that contain lines. These hierarchies may be entirely or partially independent of each other, so that a paragraph may begin on one page and end on the next, a page may contain an entire paragraph, and a paragraph may contain an entire page. Whether we consider the informational or the physical hierarchy in the present example our primary object of study may depend as much on our research agenda as it does on the text itself. Furthermore, once we have identified the hierarchy we wish to study, we do not need to examine every fine-grained detail with fractal precision, and we can concentrate instead on just the features that we actually care about for our specific purposes. For example, bibliographic entries can be very richly structured: they tend to consist of authors, titles, publishers, years of publication, etc. If we need to access these subcomponents quickly, it makes sense to encode the hierarchical structure of an individual bibliographic entry. If all we intend to do with our bibliographic entries is read them, we can treat each of them like a little paragraph, ignoring the internal structure.

Every digital humanities project begins with *document analysis*, where we determine those aspects of the inherent hierarchical structure(s) of our documents that we wish to use for research. If we are able to identify a useful hierarchical structure, the next step is to *mark up* or *encode* our document to make that hierarchy accessible to a computer. If we simply pass the computer an indifferented stream of text, it will not be able to identify the beginning and end of the various structural subcomponents. Markup is the process of inserting information into our document (this is called *tagging* in the XML context) that will take the structure that humans recognize easily and make it accessible to a computer.

Types of markup

The markup used in digital humanities projects is descriptive, which means that it describes what a textual subcomponent is. Descriptive markup differs from presentational markup, which describes what text looks like. For example, presentational markup might say that a sequence of words is rendered in italics, without any explanation of whether that's because they're a book title, a foreign phrase, something intended to be emphasized, etc. Descriptive markup also differs from procedural markup, which describes what to do with text (e.g., an instruction to a word processor to switch fonts in a particular place). The rationale for privileging descriptive markup over presentational and procedural markup in digital humanities scholarship is that descriptive markup can be transformed into presentational or procedural markup when needed for a particular purpose, but the reverse is not the case. For example, if foreign words and book titles are both marked up only as italic, it is not possible for a computer to tell whether a particular italic moment is foreign, a title, or both. If they are marked up differently, though, according to what they are rather than how they look, they can all be rendered the same way for reading purposes without losing an internal record of their difference that can be used by the computer for other purposes. The concern for encoding distinctions that may be needed for some purposes but not others is based on the idea of *multipurposing*: the user should be able to create and mark up a text only once and then use it for multiple purposes. Descriptive markup facilitates multipurposing in a way that is not the case with presentational or procedural markup.

Is every document really a hierarchy?

Would that it were that simple! Many human documents can be conceptualized (and therefore marked up in XML, in a way that will be explained below) as hierarchies, but upon close examination many turn out to have
Could I please see some XML already?

Here is a sample XML document:

```xml
<shopping_list>
  <item>bread</item>
  <item>milk</item>
</shopping_list>
```

XML models the hierarchy of a document by using elements, such as `shopping_list` and `item`. An element consists of a start tag, an end tag, and content, which is whatever occurs between the tags. A start tag is delimited by angle brackets and an end tag looks like a start tag except that it has a forward slash after the opening angle bracket. The name of the element (e.g., `shopping_list`, `item`) is technically called a generic identifier (or gi).

Don't confuse elements with tags; an element consists of the start tag, the end tag, and everything in between, which may be other elements, plain text, or a mixture of the two (see below). The tags are actually written into the XML document during editing, and XML-aware software is then capable of using that information when needed and suppressing it when it is unwanted. For example, when XML documents are presented to end users, the end users typically don't see the angle brackets. What the users might see instead, though, is that the shopping list above has been numbered automatically because the system that renders the XML for human consumption knows to insert the numbers while hiding the angle brackets and the names of the elements.

How the XML is transformed before being shown to the user is entirely under the control of the developer, and it is this control that enables the developer to encode a document solely according to what it means, without regard to how it should be styled when it is eventually presented to the reader. Many advantages of this separation of meaning and presentation are obvious. For example, if you write numbers into an ordered list and then insert, delete, or move an item while editing, you have to change the numbers, which is not only tedious and fragile, but also unnecessary. The numbering in an ordered list, after all, is actually just a rendering of the fact that the items occur in a logical order, and since that order can be determined by counting them, there is no need to enter the numbers yourself just to get them to appear in the output. The separation of content from presentation underlying XML means that by putting the items in order you’ve already made it possible for the computer to count them and insert the numbers during rendering. That the list is a sequence of items is the meaning, but the fact that that sequences may be numbered or lettered or bulleted or separated by commas in running text or otherwise formatted is a presentational decision that doesn’t change the meaning: a list is still a list, no matter how it is presented.

In XML the person who creates the document decides what to call the elements and where to use them. In the example above, the document consists of a `shopping_list` element, which contains two `item` elements. This models our understanding of what a shopping list really is: it’s a list of items to buy. This isn’t the only way to model a shopping list, though. We might, for example, organize our own lists according to sections of the market, in which case we would have another level of structure between the root `shopping_list` element and the individual `item` elements. How we choose to model our shopping list is part of document analysis; we determine
the hierarchy that interests us and introduce markup into our document to make that hierarchy accessible to a computer.

**Types of element content**

XML elements may have four types of content:

1. *Element content*. An element may contain only other elements. For example, a *shopping_list* element may contain nothing other than *item* elements.
2. *Text content*. An element may contain only plain text. For example, an *item* element might contain just the name of the item, with no other elements.
3. *Mixed content*. An element may contain a mixture of plain text and other elements. This is very common in digital humanities projects. For example, a paragraph may contain mostly plain text, but the researcher may have tagged certain words or phrases because they are structurally or semantically important. For example, a place name may be tagged as a *place* element, along the lines of:

   ```xml
   <paragraph>The American writer Jack London never lived in <place>London</place>. </paragraph>
   ```

   An author might wish to create an index of place names for the document, or cause a map to appear when the reader mouses over a place name while reading, or make it possible to search for the string “London” when it refers to the place, but not when it refers to the writer. Marking up place names differently from personal names facilitates all of these actions.
4. An element may have no content, in which case it is an *empty element*. Empty elements may be used to mark moments in a document that have no associated text. For example, a reader might want to insert a bookmark into a document to indicate a particular point, or milestone, that is not associated with any span of text.

**Attributes**

Elements may also incorporate *attributes*, which are additional markup that provides supplementary information about an element. For example, instead of just tagging a foreign word as foreign, one might wish to distinguish French words in an English text from German words in the same text. One could create *French* and *German* elements, but then there would be no easy way to formalize the fact that both elements would be used to identify foreign words. Attributes make it possible to say something like “the textual content of this element is foreign, but it is also specifically French, which means it is similar to German text insofar as both are marked up as foreign, but different because it uses a different foreign language.”

Attributes are written inside the start tag of an element (but not inside the end tag), and they consist of an attribute *name* and an attribute *value*. The attribute value must be enclosed in either double straight quotation marks (“”) or single straight apostrophes (’). Double quotation marks and single apostrophes function identically, but the marks on either side of the attribute value must agree. For example, a French word might be marked up as `<foreign language="french">oui</foreign>` and a German word as `<foreign language="german">ja</foreign>`. This markup makes it possible to treat the two similarly (based on their shared generic identifier) or differently (based on their different values of the *language* attribute).

**The XML tree and its serialization**

Marking up a document in XML consists of performing document analysis (determining the structural hierarchy and the semantics) and inserting tags (markup) into the text. XML may look like a sequence of characters, some of which are data content and some of which are markup, but structurally it is a tree, and the stream of characters, with markup and data content mixed together, is just the way it is presented to humans for reading. This representation of the XML tree (what it really is) as a stream of characters is called a *serialization*. 

63
The distinction between the tree and the serialization is important because XML trees have certain properties that must be maintained in the serialization. These are

- The XML tree has a single root, that is, a single element that contains all other elements.
- All elements must be properly nested. This means that if you open element X and inside that you open element Y, you must close element Y before you close element X. That is, element Y must nest entirely within element X. It cannot start inside but close outside or start outside and close inside.

XML documents that meet the preceding two requirements are called well formed, and all XML documents must be well formed. (There are a few additional small and self-explanatory well-formedness requirements, such as wrapping attribute values in apostrophes or quotation marks, and these will be mentioned in the course as they arise.) The following is not well formed and therefore it is not an XML document, even though it consists of a mixture of text and markup:

```xml
<dairy>
  <item>milk</item>
  <item>cheese</item>
</dairy>
<snacks>
  <item>potato chips</item>
  <item>peanuts</item>
</snacks>
```

The preceding is not well formed because it doesn't have a single root element that contains everything else. To change it into well-formed XML, wrap the dairy and snack elements in a root element, such as shopping_list. The following also is not well formed:

```xml
<paragraph>He responded emphatically in French: <emph><foreign language="french">oui</foreign></emph>!</paragraph>
```

This example has a single root element (paragraph), but the emph and foreign elements inside the paragraph element are not properly nested. The word oui is both emphatic (marked with the emph element) and French (tagged as foreign, with the attribute language used to specify the exact foreign language), and it might appear as if the internal markup begins by identifying it as emphatic and foreign, then gives the word, and then indicates that whatever follows is no longer emphatic or foreign. That appearance is deceiving; the problem is that tags are not toggle switches, which is to say that if you open the emph element and then the foreign element, you must then close the foreign element before you close the emph element, so that the foreign element will be entirely nested (start tag, contents, and end tag) inside the emph element. In the erroneous example above, the elements are not nested because emph does not contain foreign and foreign does not contain emph; instead, the tags cross.

If XML were a stream of characters, the order in which one turns on and off the emph or foreign properties might not matter. The problem is that XML only looks like a stream of characters because that's how humans read text. As was noted above, that stream of characters is just a serialization of what is logically and structurally a hierarchy. The relative order of the end tags may not matter to the way humans understand the meaning, but they matter to an XML processor. If your elements are not properly nested, what you have isn't XML.

Every new XML user makes this mistake, and for many the concept is difficult to grasp, so it bears repeating:

1. **XML is a hierarchical tree. It is not a stream of characters.** It may look like a stream of characters, but that's just the way it is presented to the human user. Internally it is a hierarchy of elements (not tags), which means that each element (start tag, contents, and end tag) must nest fully inside other elements. The only exception is the root, which, as the outmost element that contains the entire rest of the document, logically cannot nest within any other element.
2. **Tags are not toggle switches that turn properties on and off.** Tags are the way elements are serialized by inserting angle brackets and other text into data content, but XML is really a hierarchy of elements and the tags are just a way of representing that hierarchy in a linear (serial) fashion. When you open an element, you need to close it, and when you open it in a particular context (for example, inside another element), you need to close it before you can leave that context.

3. **When you’re writing XML, insert the whole element, with both the start and end tags, at the same time and only then back up and fill in the content.** This will ensure that you think in terms of nesting elements inside other elements, and not in terms of tags. When you are adding markup to existing plain-text documents to convert them to XML, don’t just drop a start tag in one place and an end tag in another. If you use an XML editor (like the `<oxygen/>` editor we will use in this course), it lets you select the text you want to mark up and wrap it in start and end tags, inserting both simultaneously. This will help you think of the element hierarchy instead of thinking about the start and end tags separately.

### Valid and well-formed documents

As was noted above, all XML documents *must* be *well formed*, which means that they must have a single root element that contains all other elements, and all elements must be properly nested, with no overlapping tags. If it isn’t well formed, it isn’t XML.

Additionally, XML *may* be *valid*. This is a technical term that means that the document uses only certain elements, and that it uses them only in certain contexts. For example, a dictionary entry might contain a head word, a pronunciation, and a series of meanings, each with examples. If you want those component elements always to occur in the same order, and you want to ensure that, for example, a single head word can be associated with multiple definitions, each of which can have multiple illustrative examples, you can write a set of rules determining where the various elements can and cannot occur and which ones can or cannot be repeated. This set of rules, or *document grammar*, is called a *schema*, and XML provides several *schema languages* that are capable of formalizing a document grammar. An XML document that is well formed can be *validated* against a schema to determine whether it follows the rules, and this is commonly done during authoring and markup to ensure that one has used the elements and attributes in a consistent way. It is possible for a document to be well formed but invalid. It is not possible for a document to be valid but not well formed, since if it isn’t well formed, it isn’t an XML document, and therefore cannot logically be a valid XML document.

All XML documents must be well formed, but whether they need to be valid (and need to be validated) depends on how you plan to use them. In practice, almost all digital humanities projects validate documents against a schema designed with the project goals in mind. In this course you will learn how to write a document grammar in a schema language and validate a document against it.

Some XML that is not well formed may nonetheless be *well balanced*. A well-balanced document fragment is one that may lack a single root node, but is otherwise well formed, that is, that does not have overlapping or missing tags. All well-formed documents are also well balanced (since not having overlapping or missing tags is one of the two requirements for well-formedness and the only requirement for being well balanced), but the reverse is not necessarily true. An XML *document* cannot be merely well balanced; if it isn’t well formed, it isn’t an XML document. An XML *document fragment*, though, may be merely well balanced, and there are stages in a digital humanities project where you may need to create well-balanced XML fragments that you will insert into an XML document, producing a well-formed (and perhaps also valid) XML document at the end. We’ll discuss the use of well-balanced XML fragments later in the course.

### Entities and numerical character references

An XML document uses angle brackets to delimit markup. For this reason, XML cannot contain an angle bracket that is meant to represent a textual character, since XML software would be unable to distinguish this textual data from markup. XML reserves two characters that cannot be represented directly in text: the left angle bracket or less-than sign (“<”) and the ampersand (“&”). When these characters occur in text that is to be represented in XML, they must be replaced in the underlying marked-up document by *entities*. Entities begin
with an ampersand and end with a semicolon; the part in between identifies the meaning of the entity. If you need to include these two characters as textual data characters in your XML, you can represent the left angle bracket as \&lt; (the letters stand for “less than”) and the ampersand as \&amp;. For example, to write “1 < 2” in an XML document you can write 1 \&lt; 2. If you write 1 < 2 literally, an XML application will be unable to parse your document, that is, unable to read it and understand its hierarchical structure.

These alternative representations are called character entities and they are essentially the only way to represent the characters in question in XML. XML also recognizes three other character entities: \&gt; (for “>”), \&quot; (for a double straight quotation mark, “ ””), and \&apos; (for a straight apostrophe, “ ’ ”). These three (unlike the first two) can usually also be entered as plain text, but there are situations where their use is restricted, and the additional entities are designed to cope with those situations. For what it’s worth, I usually use an entity for the right angle bracket and plain text for the quotation mark and apostrophe. When you undertake a digital humanities project using a text that was created outside of an XML environment, and that therefore may contain raw left angle bracket and ampersand characters, one of the first things to do is convert those to entities using a global search-and-replace operation.

Numerical character references look somewhat like entities in that they begin with an ampersand and end with a semicolon, but the opening ampersand must be followed by a hash mark (“#”) and then a number (e.g., \&#xa;). The hash mark in the preceding example indicates that we’re dealing with a numerical character reference, and not a character entity. The letter x tells us that the number is hexadecimal (base sixteen; if you don’t know what this means, don’t worry about it for now). A regular decimal (base ten) number could have been used instead; hexadecimal \&#xa; and decimal \&#160; mean the same thing, i.e., represent the same character value. The meaning of specific numbers (that is, the characters they represent) is based on Unicode, about which see below.

Numerical character references are particularly useful for representing characters that are difficult to display. For example, a web browser is able to distinguish a plain space from a non-breaking space, and it knows not to wrap to a new line between two words that are separated by only a non-breaking space. This is handy when you need to ensure that certain words will be kept together even when the user resizes the browser window. The regular and the non-breaking space differ in their behavior, but not their appearance, which means that it would be impossible to distinguish between them visually if all you could do was type the raw character. On the other hand, it is easy to distinguish a plain space character (regular space) from the string \&#xa;, which is the numerical character reference corresponding to the non-breaking space character. Any character may be represented by a numerical character reference, but normal practice is to write raw characters (normal letters, numbers, etc.) unless there is a good reason not to (as with different types of spaces, which look alike to the human eye).

Since whether a character is represented as a raw character or a numerical character reference has no effect on the meaning (the representations are exactly equivalent informationally), that choice is part of the serialization that the human reads, rather than the internal representation that the XML processor sees. This means that when you write a program to read your XML and convert it to different XML, it may or may not replace raw characters with numerical character references or vice versa. There are limited ways to control this aspect of the serialization while processing XML, but normally you shouldn’t worry about it, since it has no effect on meaning, and is not considered informational in an XML context.

**The life cycle of an XML document**

When you undertake a digital humanities project, you’ll work with one or more documents. Those may already exist as XML documents (e.g., you might use XML that someone else created for a different purpose), or they may already exist as documents in other forms. (In that latter case, you’ll need to convert them to XML, which typically involves a mixture of auto-tagging, where you run some global search-and-replace operations to insert markup, and manual tagging.) Or you may create new documents entirely from scratch, where you are creating not just the markup, but also the data content. Whatever the source of your data:

1. The first stage of a digital humanities project is document analysis, where you determine the hierarchical structure of the documents that you wish to model in XML.
2. You then typically begin to design a schema, a formal grammar describing where the various elements and attributes you want to use can and cannot be employed.

3. Once you have a draft schema, you begin to mark up your document according to the schema and validate it, that is, verify that you have used markup only in a way that conforms to the schema.

The preceding three steps constitute an iterative process. It is very common during the markup phase to notice that the document analysis or the schema needs to be refined and modified, and once you've changed the schema, you often need to adjust the markup accordingly. In an ideal world, you would complete each of the three steps before beginning the next, so that you wouldn't have to redo or undo previous work, but in practice, no matter how scrupulous the data analysis, there are almost inevitably complications that become evident only during the markup phase. Nonetheless, because a lot of structural and semantic information inherent in your documents will easily be discernible during document analysis, and getting your schema started before beginning the markup leads to better markup, it is never a good idea to skip the document-analysis and schema-development stages completely and begin with marking up text.

Once your documents have been encoded, you'll write programs in XML-oriented languages to transform them, publish them (often in very different views, with different selections and arrangements of the same underlying data), analyze them, and generate reports. Because what you choose to model in your schema and mark up in your documents is dictated by your research agenda, it is important to conduct your document analysis and develop your schema with your goals in mind. For example, if you know that you want to study issues of gender in Shakespeare’s plays, you need to think at the beginning of how your programs will identify gender-related features, and those considerations typically inform your schema design and markup. Don't try to mark up a text until you've given some thought to what you want to do with it, that is, to why you want to mark it up in the first place.

The XML family of standards

XML is part of a family of standards that cooperate to support the study and processing of texts. We'll learn about all of these standards in this course, and we'll use some of them in our own work. Don't worry about memorizing them now because they'll become familiar over the semester, but so that you'll have all of the names in one place, they include:

Schema languages

Languages used for the formal modeling of document structure. The schema languages in use in the XML world are Document Type Definitions (DTDs), Relax NG, and W3C Schema. In this course we use Relax NG.

XSLT

eXtensible Style Sheet Language Transformations, a programming language used to transform XML to other forms (other XML, HTML [web pages], plain text, etc.). The term style sheet is somewhat misleading; XSLT can be used to style an XML document, but it can also be used to create entirely new documents by transforming existing ones in almost unlimited ways.

XQuery

A language used to query XML databases.

XPath

A formal method of navigating the XML hierarchy, used by XSLT and XQuery. For example, if you want to generate a table of contents based on chapter titles, you use XPath to find (navigate to) all the chapter titles in your document and XSLT to generate an output document that contains the newly constructed table of contents.

XSL-FO

eXtensible Stylesheet Language Formatting Objects, used with XSLT to transform XML into Portable Document Format (PDF, the type of document read in Adobe Reader).
SVG
Scalable Vector Graphics. An XML vocabulary (schema) for describing graphics. Useful in digital humanities projects for creating graphic representations of textual data. For example, one could use XSLT to transform a Shakespearean play into a bar graph illustrating how much each character speaks.

Namespaces
A technology used to manage different XML vocabularies in the same project. For example, if your project includes a schema with elements for bibliography, which might include a title element for book titles, together with a schema for representing persons, which might also include a title element, this time for royalty, namespaces allow you to call them both title in your document while retaining the ability to distinguish them.

Schematron
A constraint modeling language used to restrict what is permitted in a document in ways that schema languages alone cannot address.

XProc
A pipelining language, intended to allow the user to transform one XML document into another in stages, feeding (“piping”) the output of one transformation into the next as input.

Regular expressions
A symbolic system for representing text with wildcards or in other flexible ways. Regular expression processing might be a part of a routine to convert a date in “4/1/2011” (month/date/year) format to something like “April 1, 2011.” A regular expression would be used to break the string of digits and slashes into the three subcomponents of the date, and they could then be stitched back together differently, looking up the name of the month in a table and inserting punctuation and spaces where needed.

XForms
An XML-oriented replacement for the forms used on the Web on commercial sites and elsewhere.

Other web standards
XML can be used without reference to the Web or the Internet. For example, one could write XSLT to generate a table of information from an XML document and copy and insert it into a Microsoft Word document for printing on paper—all without ever being connected to the Internet. Nonetheless, XML is most commonly used in a Web-aware environment, where it interacts with the following standards:

HTML
Hypertext Markup Language is a set of XML schemas used to describe the structure of web pages. There are several versions and variants of HTML; in this course we will most commonly be transforming our XML into XHTML 1.1 for publication on the Web.

CSS
Cascading Style Sheets are a strategy for describing how XML (and HTML) should be rendered. Descriptive markup describes what the elements in a document mean, but not how they look, and CSS is intended to let the designer specify the rendering separately from the XML, so that meaning and appearance do not become conflated or confused.

Javascript
Javascript is a client-side programming language for manipulating, among other things, the appearance of web pages in the browser. Client-side means that Javascript runs in the user’s browser, so that, for example, the user can change what is rendered in the browser window without having to fetch new information from the server (the place from which the original page was downloaded). Javascript is frequently used to modify CSS in response to user activities or events, such as mouse clicks or movements or keyboard presses, so that the
document remains the same inside the browser, but the way it is rendered on the screen is updated in response to events. The use of Javascript to modify CSS on the client side is called dynamic HTML (DHTML). One very common use of Javascript involves the collapsing menus that proliferate on the web, where different topics expand to show subtopics and then contract to hide them in response to mouse clicks. The different views of the menu are not downloaded individually from the server. The entire menu is always present on the user’s machine, but certain parts are shown or hidden by a Javascript program that runs in the background and listens for mouse clicks.

**PHP**

PHP is a server-side scripting language, which means that it can be used to construct the web page that it sends the user after taking into consideration user-supplied or other information. PHP is commonly used for login validation; the user supplies a username and password and the system decides what to return after authenticating the user in a database. For example, the server might return a page content customized for different people according to the way their roles in an organization are stored in the database. Client-side Javascript wouldn’t work for this purpose because it would be a security violation to download the the entire password database to the user’s machine and authenticate the user there! On the other hand, because the transfer of information between a server and client can take a very long time (in computer terms), tasks normally should be performed on the client unless there is some reason they require the involvement of the server.

**Character sets and Unicode**

What the user sees on the screen as letters of a particular alphabet or numbers or punctuation marks or other symbols represent characters, the abstract units of information used to represent, among other things, written language. While humans regard letters and numbers and punctuation as fundamentally different from one another, all characters are represented inside the computer as numerical values, and those values are associated not only with how a character looks, but also with what it means. For example, the Latin-alphabet small letter “a” has the internal numerical representation of decimal 97. The Cyrillic small letter “а” looks exactly the same, but it has the internal numerical representation of decimal 1072. If all we did with text was read it and print it the difference wouldn’t matter, since we can’t see a difference just by looking, but when we search for English “а” we don’t want also to retrieve Russian “а”.

Unicode is the system all modern computers and computer software use to represent characters. You don’t usually need to think about it, and when you do, you can look up the values conveniently on the Unicode Consortium web site.

**Practicing what we preach**

This document was authored in XHTML 1.1, which is an XML schema that determines which elements may be used and how they may be used to create pages that web browsers will be able to render in conformity with user expectations. Because this page is an XML document, it is possible to use XSLT to transform it into a different XML document. As an exercise to demonstrate the use of XSLT to transform XML to different XML, before I inserted the table below into this page, I ran the page through an XSLT program, which created the table. That is, I didn’t type anything myself to write the table; instead, I wrote an XSLT program to generate it automatically from this page of prose.

To do that, in the document I had already marked up new terms with the XHTML dfn element, which is intended to identify terms that are being defined. How did I know that this was the best element to use for definition terms? Because XHTML is someone else’s schema, the elements and their use have been predefined, and I had to read the XHTML documentation (at, for example, SitePoint, an excellent general resource for web-related information). When you create your own schema, you determine the elements and the rules for their use, but when you use someone else’s, you need to read their documentation.

By default, text that is tagged as dfn is typically rendered in the browser in an oblique (slanted) or italic font, and the latter styling is also the default for text tagged as emphatic (em) or a citation (cite). In practice, of the
Because I marked up my terms according to what they are, instead of according to how they appear, my document rigorously distinguishes terms from other elements that might also be rendered in italics. For example, XHTML includes the \texttt{cite} element, intended for cited references (such as book titles), the \texttt{em} element, intended for emphasized text, and the \texttt{i} element, intended for italic text of other types. (The \texttt{i} element is a leftover from an earlier era, when HTML markup was more presentational. Although it is still valid in XHTML, many modern developers avoid it precisely because it conveys no structural or semantic information, and merely describes how to render the text visually.) Had I simply encoded all information that I wanted to render in italics with the same markup, I would not have been able to distinguish among definition terms, cited references, emphasis, and generic italic text programmatically. But because definition terms and only definition terms are marked up as \texttt{dfn} elements in my document, my XSLT program can find them easily.

To demonstrate the use of XSLT to transform XML, I wrote a \texttt{script} (another term for a program) that finds each word or phrase that has been tagged as a \texttt{dfn} element and outputs it in a table next to the header of the section in which it appears. The resulting table looks like this:

<table>
<thead>
<tr>
<th>Term (\texttt{dfn})</th>
<th>Section heading (\texttt{h2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>Attributes</td>
</tr>
<tr>
<td>auto-tagging</td>
<td>The life cycle of an XML document</td>
</tr>
<tr>
<td>character entities</td>
<td>Entities and numerical character references</td>
</tr>
<tr>
<td>characters</td>
<td>Character sets and Unicode</td>
</tr>
<tr>
<td>client-side</td>
<td>Other web standards</td>
</tr>
<tr>
<td>content</td>
<td>Could I please see some XML already?</td>
</tr>
<tr>
<td>descriptive</td>
<td>Types of markup</td>
</tr>
<tr>
<td>digital humanities</td>
<td>The context</td>
</tr>
<tr>
<td>document analysis</td>
<td>The beginning of every digital humanities project: Document analysis</td>
</tr>
<tr>
<td>document grammar</td>
<td>Valid and well-formed documents</td>
</tr>
<tr>
<td>dynamic html</td>
<td>Other web standards</td>
</tr>
<tr>
<td>element content</td>
<td>Types of element content</td>
</tr>
<tr>
<td>elements</td>
<td>Could I please see some XML already?</td>
</tr>
<tr>
<td>empty element</td>
<td>Types of element content</td>
</tr>
<tr>
<td>encode</td>
<td>The beginning of every digital humanities project: Document analysis</td>
</tr>
<tr>
<td>end tag</td>
<td>Could I please see some XML already?</td>
</tr>
<tr>
<td>entities</td>
<td>Entities and numerical character references</td>
</tr>
</tbody>
</table>
The XSLT script begins by finding the **`dfn`** elements in the page. To the XSLT processor, XML is a tree, so what the processor sees is something like the following diagram (if you don’t see a tree diagram, you’re using an old browser; upgrade to the latest version of Firefox, Internet Explorer, Chrome, Safari, or Opera):
Could I please see some XML already?

The human developer looking at the preceding XML snippet in an editor would see something like:

```xml
<body>
  <div> ... </div>
  <div>
    <h3>Could I please see some XML already?</h3>
    <p> ... </p>
    <p> ... a <dfn>start tag</dfn>, an <dfn>end tag</dfn>, and ... </p>
  </div>
  <div> ... </div>
</body>
```

The textual representation with the angle brackets is a serialization, but it isn’t the real XML. The real XML is the tree, and that, rather than the serialization, is all that’s accessible to an XML tool (such as the XSLT processor that created the table above). The XSLT script finds all of the `dfn` elements in the tree; the diagram above is a snippet of the full tree and contains two such elements (colored green). The script sorts the text of all of the terms it finds (in the blue text boxes under the green `dfn` nodes) and creates a new row in the table for each term, writing it (converted to lower case, if necessary) into the left column of the row. The script must then find the title of the section in which the term occurs and put that title into the corresponding right column of the row, and to do that it must walk the tree, starting at each `dfn` node and ending at the appropriate section header.

To find the appropriate section head, for each `dfn` node the script walks up the tree looking for the ancestral section (`div`) node. That route is colored pink and marked with arrowheads in the diagram. The script first walks up from the `dfn` to the `p` that contains it, and then from the `p` to the `div` that contains it. When the process reaches a `div`, it changes direction and looks under the `div` for the section title (`h3`) node, and it retrieves the section title text from inside the `h3` node. The text that gets copied into the table is colored blue; the text under the `dfn` node goes in the left column and the text under the `h3` goes in the right. The snippet represented by the tree creates two rows in the table, one for each `dfn` node, and since they are located inside the same `div` they are matched with the same `h3` text. The element nodes that are not used to create the table and the text that is not copied into the table are colored yellow.
All of this takes just a few lines of XSLT code. It does require me to know the structure of my input document (this page), that is, to know that I used `h3` elements for section titles and not something else. As you develop your sites, you’ll want to develop your schemas with an eye toward the processing you want to do with your data, and you’ll want to design your XSLT or XQuery programs to take advantage of the features you build into your schema.

Remember multipurposing, the programmatic generation of multiple output documents from a single XML source document? Suppose we also need to produce a completely different table, one that lists each of the section headings in the order in which it occurs in the document alongside a count of the terms that are introduced in that section and a list of the terms themselves. Oh, and let’s sort the terms alphabetically, instead of listing them in the order in which they occur, and render them all in lower case, even if they might originally have had initial capitalization. A different small XSLT program produces the following table from the same input document, this page, here too without my having to construct any part of the table manually:

<table>
<thead>
<tr>
<th>Section title</th>
<th>Count</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>The context</td>
<td>2</td>
<td>digital humanities, humanities computing</td>
</tr>
<tr>
<td>Document and their structures</td>
<td>3</td>
<td>nodes, root, tree</td>
</tr>
<tr>
<td>The beginning of every digital humanities project: Document analysis</td>
<td>4</td>
<td>document analysis, encode, mark up, tagging</td>
</tr>
<tr>
<td>Types of markup</td>
<td>4</td>
<td>descriptive, multipurposing, presentational, procedural</td>
</tr>
<tr>
<td>Is every document really a hierarchy?</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Could I please see some XML already?</td>
<td>6</td>
<td>content, elements, end tag, generic identifier, gi, start tag</td>
</tr>
<tr>
<td>Types of element content</td>
<td>5</td>
<td>element content, empty element, milestone, mixed content, text content</td>
</tr>
<tr>
<td>Attributes</td>
<td>3</td>
<td>attributes, name, value</td>
</tr>
<tr>
<td>The XML tree and its serialization</td>
<td>3</td>
<td>serialization, serialization, well formed</td>
</tr>
<tr>
<td>Valid and well-formed documents</td>
<td>6</td>
<td>document grammar, schema, schema languages, valid, validated, well balanced</td>
</tr>
<tr>
<td>Entities and numerical character references</td>
<td>3</td>
<td>character entities, entities, numerical character references</td>
</tr>
<tr>
<td>The life cycle of an XML document</td>
<td>1</td>
<td>auto-tagging</td>
</tr>
<tr>
<td>The XML family of standards</td>
<td>1</td>
<td>style sheet</td>
</tr>
<tr>
<td>Other web standards</td>
<td>4</td>
<td>client-side, dynamic html, events, server-side</td>
</tr>
<tr>
<td>Character sets and Unicode</td>
<td>1</td>
<td>characters</td>
</tr>
<tr>
<td>Practicing what we preach</td>
<td>1</td>
<td>script</td>
</tr>
<tr>
<td>Conclusion</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

This is a modest example of the use of XSLT to transform XML, constrained by my desire to work with the present document. In this course you’ll identify more interesting processing tasks, of the sort that can be used to facilitate real humanities research, and you’ll design systems and write programs to undertake that research.

**Conclusion**
The most important things to remember from this introduction are that:

- XML is a way of modeling a textual document as an ordered hierarchy, or tree, so that it can be explored with computational tools. Humanities scholars use XML to represent their documents because the tree model is convenient both as a logical representation (some aspects of the inherent structure of documents are tree-like) and for programming purposes (computers can process tree representations very efficiently).
- Digital humanities researchers typically mark up the documents they wish to study in XML and then transform, explore, and manipulate those documents using XSLT (and XQuery).
- XML in digital humanities research should be used to model the structure and semantics of a document. Presentation is important, but it should be handled separately, and not confused or conflated with structure and semantics.
- The life cycle of a digital humanities project starts with document analysis and also includes schema design, markup, and processing.
- XML markup consists of elements and attributes. Sometimes unusual written symbols (characters) may be represented more easily by character entities or numerical character references than as raw text.
- XML is a tree of elements. It is not a stream of tags and text. It looks like a stream because we usually read character by character, but the internal representation of an XML document, which is what the computer sees, is an ordered hierarchy of objects, and not a string of characters.
- XML must be well formed. It may also be validated against a particular schema. In digital humanities projects XML is usually validated against a schema.

It isn’t necessary to memorize—or even understand—all of the details of this introduction on first reading. You probably understood some parts very well, while others may have seemed complex or confusing. That’s not a problem; you’ve been exposed to the terms, which will sound familiar and make more sense when you hear them again in context, and you can return to this introduction at any time to as you gain more hands-on experience working with XML yourself.
Introduction

Julia Fransen, Syd Baumann
The focus of XML: nest, attributes, and understanding.

TECHNICAL ORIENTATION: UNDERSTANDING

attributes

The basis of XML: nest, attributes, and understanding.

XML...
In which it is stated:

1. Elements may overlap each other and occupy the same physical space.
2. Attributes, such as "class" or "id", are stored on the element, not in the surrounding text.
3. Text flows around the element, not through it.
4. Elements are defined by tags, which are used to create documents.

You have now learned the essentials of XML, which apply to all XML applications.

To import XML into a document or program, you need to know the syntax of the XML file you are working with. XML is a language for describing a document's structure, not its content. It is used to store and transport large sets of data, such as web pages or email messages.

The rules of XML:

- **Well-formedness and validity:**
  - Well-formedness means that the document is syntactically correct.
  - Validity means that the document is syntactically correct and conforms to a DTD or schema.

**Pointing and linking:**

- Elements can point to other elements in a document or in other documents.
- Attributes can be used to specify links to other elements or documents.
- URLs can be stored in attributes, allowing for easy linking to other resources.

**Contracts and coordination:**

- In XML, it is possible to create contracts between elements or documents.
- These contracts can be used to specify how elements should behave or interact with each other.
- Contracts are enforced through the use of rules defined in a DTD or schema.

**Hierarchical relationships:**

- Elements are arranged in a hierarchical manner, with parent and child relationships.
- Each element can have multiple children, and each child can have multiple parents.

**Automatic generation from data:**

- XML can be used to automatically generate data from other sources, such as databases or spreadsheets.
- This is done through the use of XSLT (Extensible Stylesheet Language Transformations).

**Scripting and interaction:**

- XML can be used to create dynamic web applications, allowing for user interaction.
- This is done through the use of JavaScript or other scripting languages.

**Security and access control:**

- XML can be used to create secure web applications, allowing for restricted access to certain elements or documents.
- This is done through the use of XACML (eXtensible Access Control Markup Language).

**Data interchange:**

- XML can be used to interchange data between different systems or applications.
- This is done through the use of XML schemas or DTDs, which define the structure of the data.

**XML is not XML:**

- XML is not a programming language, but rather a markup language.
- It is used to describe the structure of data, not to write programs.

XML is a powerful tool for describing the structure of data, and it is widely used in a variety of applications, from web pages to scientific experiments.
Chapter 4: The Kitchen House

Let's begin with a simple example: there is a counted example of some source

addressed our cognitive attention as such.

For instance: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operating in a natural environment of its own, and the

example: a computer operat
Using the TEL

The TEL is a computer program that allows you to create and manipulate text documents. It provides a user-friendly interface for inserting, deleting, and modifying text. The TEL also supports advanced features such as spell checking, grammar checking, and document management. In this section, we will explain how to use the TEL.

1. **Opening the TEL**

   To open the TEL, you need to double-click on the TEL icon located on your desktop. This will launch the program and bring up the main window.

2. **Creating a new document**

   Once the TEL is open, you can create a new document by selecting the option "New Document" from the File menu or by clicking on the "New" button on the toolbar. This will open a new blank document where you can start typing.

3. **Entering text**

   To enter text into your document, simply click on the text area and start typing. As you type, the text will appear on the screen.

4. **Editing text**

   To edit your text, you can use the mouse to select the text you want to modify. You can then use the delete key to remove characters or the backspace key to insert new characters. You can also use the cut, copy, and paste functions to move text around.

5. **Saving your document**

   Once you have finished typing your document, you should save it to prevent losing any changes. You can do this by selecting the option "Save" from the File menu or by clicking on the "Save" button on the toolbar. This will save your document to the default location on your computer.

6. **Closing the TEL**

   To close the TEL, you can select the option "Close" from the File menu or by clicking on the "X" button in the top-right corner of the main window. This will close the program and exit.

By following these steps, you can effectively use the TEL to create and manage your text documents.
The text on the page appears to be a continuation of the previous page, discussing strategic planning for encoding projects. The content likely involves planning and development aspects related to encoding and metadata management. Here is a partial transcription:

"Strategic Planning for Encoding Projects"

This section appears to be discussing the importance of planning encoding projects, possibly including considerations for metadata, preservation, and project management. The text may be guiding the reader through the steps necessary for effective strategic planning in encoding projects.