Part IV

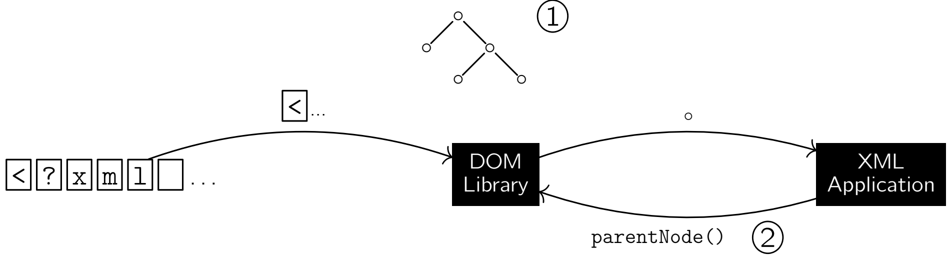
DOM—Document Object Model

Outline of this part

1. DOM Level 1 (Core)
2. DOM Example Code
3. DOM—A Memory Bottleneck

DOM—Document Object Model

- With **DOM**, W3C has defined a **language-** and **platform-neutral** view of XML documents, much like the XML Information Set.
- DOM APIs exist for a wide variety of—predominantly object-oriented—programming languages (Java, C++, C, Perl, Python, ...).
- The DOM design rests on two major concepts:
  1. An **XML Processor** offering a DOM interface parses the XML input document, and constructs the **complete XML document tree** (in-memory).
  2. The **XML application** then issues DOM library calls to **explore** and **manipulate** the XML document, or **generate** new XML documents.



- The DOM approach has some obvious advantages:
  - Once DOM has build the XML tree structure, (tricky) issues of XML grammar and syntactical specifics are void.
  - **Constructing** an XML document using the DOM instead of serializing an XML document manually (using some variation of `print`), ensures **correctness** and **well-formedness**.
    - No missing/non-matching tags, attributes never owned by attributes, ...
  - The DOM can simplify document **manipulation** considerably.
    - Consider transforming

Weather forecast (English)

```
1 <?xml version="1.0"?>
2 <forecast date="Thu, Nov 8">
3   <condition>sunny</condition>
4   <temperature unit="Celsius">-1</temperature>
5 </forecast>
```

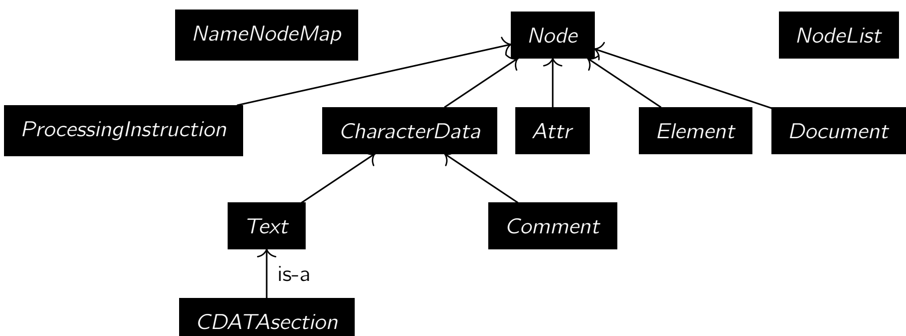
into

Weather forecast (German)

```
1 <?xml version="1.0"?>
2 <vorhersage datum="Do, 8. Nov">
3   <wetterlage>neblig</wetterlage>
4   <temperatur einheit="Celsius">-1</temperatur>
5 </vorhersage>
```

DOM Level 1 (Core)

- To operate on XML document trees, DOM Level 1[2] defines an inheritance hierarchy of node objects—and methods to operate on these—as follows (excerpt):



- Character strings (DOM type *DOMString*) are defined to be encoded using UTF-16 (*e.g.*, Java DOM represents type *DOMString* using its `String` type).

[2] `http://www.w3.org/TR/REC-DOM-Level-1/`

- (The complete DOM interface is too large to list here.) Some methods of the principal DOM types *Node* and *Document*:

DOM Type	Method		Comment
Node	*nodeName*	:: *DOMString*	redefined in subclasses, *e.g.*, tag name for *Element*, `"#text"` for *Text* nodes, ...
	parentNode	:: *Node*	
	firstChild	:: *Node*	leftmost child node
	nextSibling	:: *Node*	returns `NULL` for root element or last child or attributes
	childNodes	:: *NodeList*	see below
	attributes	:: *NameNodeMap*	see below
	ownerDocument	:: *Document*	
	replaceChild	:: *Node*	replace new for old node, returns old
Document	*createElement*	:: *Element*	creates element with given tag name
	createComment	:: *Comment*	creates comment with given content
	getElementsByTagName	:: *NodeList*	list of all *Elem* nodes in document order

Some DOM Details

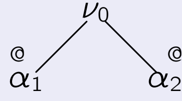
- Creating an element (or attribute) using *createElement* (*createAttribute*) does *not* wire the new node with the XML tree structure yet.

  Call *insertBefore*, *replaceChild*, ... to wire a node at an explicit position.
- DOM type *NodeList* (node sequence) makes up for the lack of collection datatypes in most programming languages.

  Methods: *length*, *item* (node at specific index position).
- DOM type *NameNodeMap* represents an *association table* (nodes may be accessed by name).

  Example:

  `bubble`

  `speaker` `to`

  Apply method *attributes* to *Element* object ν_0 to obtain this *NameNodeMap*:

name		**node**
`"speaker"`	$\mapsto$	α_1
`"to"`	$\mapsto$	α_2

  Methods: *getNamedItem*, *setNamedItem*, ...

DOM Example Code

- The following slide shows C++ code written against the Xerces C++ DOM API[3].
- The code implements a variant of the *content* :: *Doc* → (*Char*):
  - Function `collect()` decodes the UTF-16 text content returned by the DOM and prints it to standard output directly (`transcode()`, `cout`).

N.B.

- A W3C DOM node type named τ is referred to as `DOM`τ in the Xerces C++ DOM API.
- A W3C DOM property named *foo* is—in line with common object-oriented programming practice—called `getFoo()` here.

[3]`http://xerces.apache.org/xerces-c/`

Example: C++/DOM Code

content.cc (1)

```
11
12  void collect(DOMNodeList *ns)
13  {
14    DOMNode *n;
15
16    for (unsigned long i = 0;
17         i < ns->getLength();
18         i++){
19      n = ns->item(i);
20
21      switch (n->getNodeType())
22      {
23        case DOMNode::TEXT_NODE:
24          cout << XMLString::transcode(
25                        n->getNodeValue());
26          break;
27        case DOMNode::ELEMENT_NODE:
28          collect(n->getChildNodes());
29      }
30    }
31  }
```

content.cc (2)

```
32
33  void content(DOMDocument *d)
34  {
35    collect(d->getChildNodes());
36  }
37
38  int main(void)
39  {
40    XMLPlatformUtils::Initialize();
41
42    XercesDOMParser *parser =
43                       new XercesDOMParser();
44    DOMDocument *doc;
45
46    parser->parse("dilbert.xml");
47    doc = parser->getDocument();
48
49    content(doc);
50
51    return 0;
52  }
```

Now: Find all occurrences of Pointy-Haired Boss (`phb`) speaking (attribute `speaker` of element `bubble`) ...

dogbert.cc (1)

```
12  void dogbert(DOMDocument *d)
13  {
14    DOMNodeList     *bubbles;
15    DOMNode         *bubble, *speaker;
16    DOMNamedNodeMap *attrs;
17
18    bubbles = d->getElementsByTagName(XMLString::transcode("bubble"));
19
20    for (unsigned long i = 0; i < bubbles->getLength(); i++) {
21      bubble = bubbles->item(i);
22
23      attrs = bubble->getAttributes ();
24      if (attrs != NULL)
25        if ((speaker = attrs->getNamedItem(XMLString::transcode("speaker")))
26            != NULL)
27          if (XMLString::compareString(speaker->getNodeValue(),
28                                       XMLString::transcode("phb")) == 0)
29            cout << "Found Pointy-Haired Boss speaking." << endl;
30    }
31  }
```

dogbert.cc (2)

```
32 int main (void)
33 {
34   XMLPlatformUtils::Initialize ();
35
36   XercesDOMParser *parser = new XercesDOMParser();
37   DOMDocument *doc;
38
39   parser->parse("dilbert.xml");
40   doc = parser->getDocument();
41
42   dogbert(doc);
43
44   return 0;
45 }
```


DOM—A Memory Bottleneck

- The two-step processing approach (① parse and construct XML tree, ② respond to DOM property function calls) enables the DOM to be "**random access**":

  The XML application may explore and update any portion of the XML tree at any time.

- The inherent memory hunger of the DOM may lead to
  1. heavy **swapping** activity
     (partly due to unpredictable memory access patterns, `madvise()` less helpful)
     or
  2. even "`out-of-memory`" failures.
     (The application has to be extremely careful with its own memory management, the very least.)

Numbers

DOM and random node access

Even if the application touches a single element node only, the DOM API has to maintain a data structure that represents the **whole XML input document** (all sizes in kB):[4]

XML size	DOM process size DSIZ	$\frac{\texttt{DSIZ}}{\text{XML size}}$	Comment
7480	47476	6.3	(Shakespeare's works) many elements containing small text fragments
113904	552104	4.8	(Synthetic eBay data) elements containing relatively large text fragments

[4] The random access nature of the DOM makes it hard to provide a truly "lazy" API implementation.

To remedy the memory hunger of DOM-based processing . . .

- Try to **preprocess** (*i.e.*, filter) the input XML document to reduce its overall size.
  - Use an XPath/XSLT processor to preselect *interesting* document regions,
  - *no updates* to the input XML document are possible then,
  - make sure the XPath/XSLT processor is *not* implemented on top of the DOM.

Or

- Use a **completely different** approach to XML processing (→ **SAX**).