Part IV

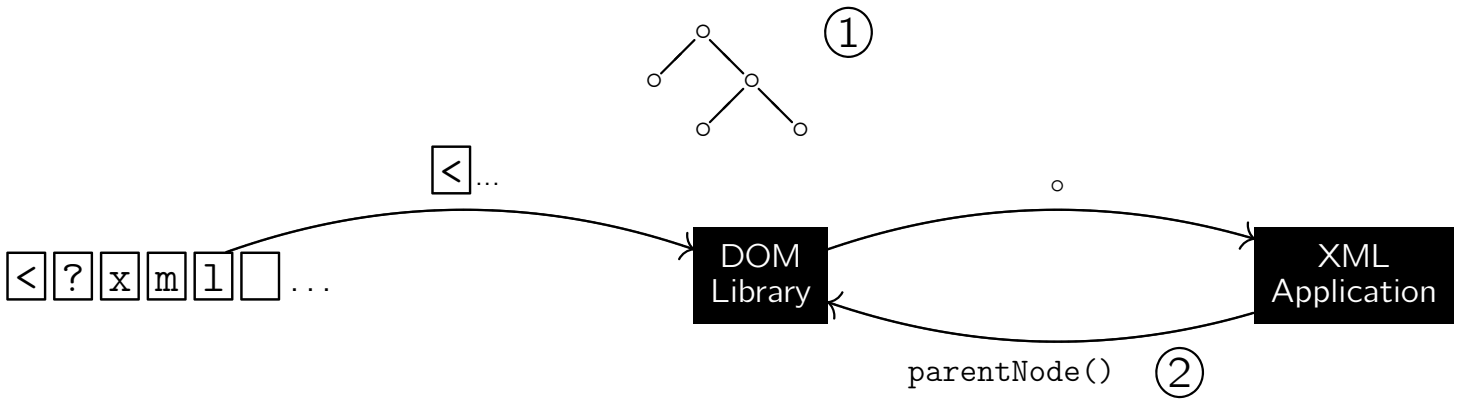
DOM—Document Object Model

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, Oct 30">
3   <condition>sunny</condition>
4   <temperature unit="Celsius">-1</temperature>
5 </forecast>
```

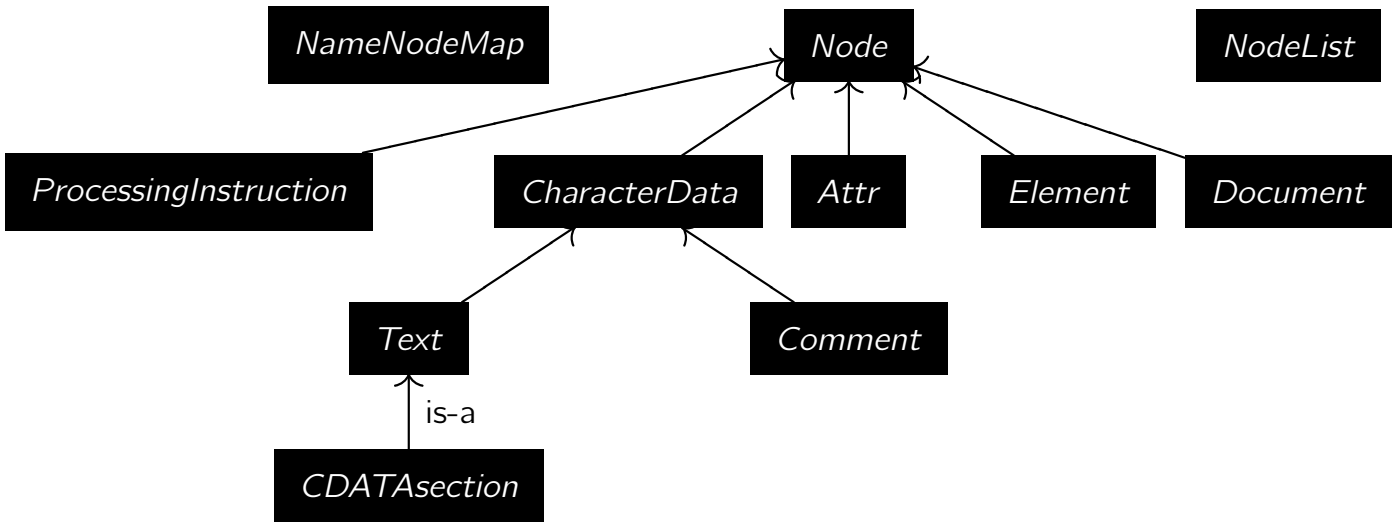
into

Weather forecast (German)

```
1 <?xml version="1.0"?>
2 <vorhersage datum="Do, 30. Okt">
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.

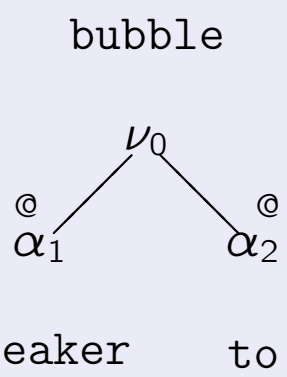
  Call *insertBefore*, *replaceChild*, ... to wire a node at an explicit position.

- 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:



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://xml.apache.org/`

Example: C++/DOM Code

content.cc (1)

```
1  // Xerces C++ DOM API support
2  #include <dom/DOM.hpp>
3  #include <parsers/DOMParser.hpp>
4
5  void collect (DOM_NodeList ns)
6  {
7    DOM_Node n;
8
9    for ( unsigned long i = 0;
10         i < ns.getLength ();
11         i++){
12     n = ns.item (i);
13
14     switch (n.getNodeType ()) {
15     case DOM_Node::TEXT_NODE:
16       cout << n.getNodeValue ().transcode ();
17       break;
18     case DOM_Node::ELEMENT_NODE:
19       collect (n.getChildNodes ());
20     }
21   }
22 }
```

content.cc (2)

```
23
24 void content (DOM_Document d)
25 {
26   collect (d.getChildNodes ());
27 }
28
29 int main (void)
30 {
31   XMLPlatformUtils::Initialize ();
32
33   DOMParser parser;
34   DOM_Document doc;
35
36   parser.parse ("foo.xml");
37   doc = parser.getDocument ();
38
39   content (doc);
40
41   return 0;
42 }
```

Now: Find all occurrences of `Dogbert` speaking (attribute `speaker` of element `bubble`) ...

dogbert.cc (1)

```
1  // Xerces C++ DOM API support
2  #include <dom/DOM.hpp>
3  #include <parsers/DOMParser.hpp>
4
5  void dogbert (DOM_Document d)
6  {
7    DOM_NodeList     bubbles;
8    DOM_Node         bubble, speaker;
9    DOM_NamedNodeMap attrs;
10
11   bubbles = d.getElementsByTagName ("bubble");
12
13   for (unsigned long i = 0; i < bubbles.getLength (); i++) {
14     bubble = bubbles.item (i);
15
16     attrs = bubble.getAttributes ();
17     if (attrs != 0)
18       if ((speaker = attrs.getNamedItem ("speaker")) != 0)
19         if (speaker.getNodeValue ().
20               compareString (DOMString ("Dogbert")) == 0)
21           cout << "Found Dogbert speaking." << endl;
22   }
23 }
```

dogbert.cc (2)

```
24
25  int main (void)
26  {
27    XMLPlatformUtils::Initialize ();
28
29    DOMParser parser;
30    DOM_Document doc;
31
32    parser.parse ("foo.xml");
33    doc = parser.getDocument ();
34
35    dogbert (doc);
36
37    return 0;
38  }
```

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 ($\rightarrow$ **SAX**).