6 A.

The following are the SAX events triggered for the give xml.

startDocument();

startElement(“student”, {(“type”, ”undergrad”)});

startElement(“name”);

characters(“John Harvard”);

endElement(“name”);

startElement(“email”);

characters(“john@harvard.edu”);

endElement(“email”);

startElement(“phone”);

endElement(“phone”);

startElement(“grades”);

startElement(“project”, {(“number”, “1”)});

characters(“137”);

endElement(“project”);

endElement(“grades”);

endElement(“student”);

endDocument();

7. A

8. A

1. As the calculations require random access through the XML Document the preferred API is DOM as DOM holds the xml content in memory at any point of the time one could access the data that is parsed by DOM.
2. SAX API would be a preferred API as it is a *“fire and forget”* type API, can make the application to be driven fast.
3. The scenario is to deal with large XML file, SAX API would be a preferred one; as DOM build as tree like structure in memory, which implies large xml needs a large memory which may not be available at every time.
4. The scenario has a use case of sorting which needs the access to previous parsed data, so DOM would be preferred one if memory is not a concern. But if the memory is strictly on lesser side the scenario requires SAX API to retrieve data and store into some appropriate data structure and carry out necessary sorting.
5. SAX API would be a preferred one, as the scenario clearly suggests at each time only it is needed to extract content of one single element in an XML document quickly.

9. A

The xml formed by the given SAX events is

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<lectures>

<lecture number=*"1"* available=*"1"*>

<date>Monday, 17 September 2007</date>

<tite>Lecture 1</tite>

<subtitle>Introduction</subtitle>

<handouts>

<handout>

<name>Slides</name>

<formats>

<format available=*"0"* type=*"PDF"* filename=*"lecture1.pdf"* />

</formats>

</handout>

</handouts>

</lecture>

</lectures>

The DOM tree of the above structure is as shown below

10 A.

element :: = STag content ETag

content :: = (element | alphanumeric | whitespace character)\*

STag :: = ‘<’ Name ‘>’

ETag :: = ‘</’ Name ’>’

Name :: = ( alphabetical )\*

alphanumeric :: = ( alphabetical | numerical )

alphabetical :: = ( ‘a’ - ‘z’ | ‘A’ - ‘Z’ )

numerical :: = ( ‘0’ - ‘9’ )

12 A.

By iteratively checking and skipping over whitespace characters from the beginning of the file (skipConsecutiveWhitespaces()), till either a non-whitespace character appears or end of file is reached.

14 A.

The code for parsing the Name from the tag is already provided by the course. The parser is continued to encounter any non-whitespace character that may encounter before touching '>'.

If encountered any non-whitespace character is encountered the parser is allowed to accumlate those character till it encounters another whitespace character or '=' sign and the accumlated string is stored as attribute name,

simillarly after reaching '=' parser skips over consective whitespace characters if any present and awaits for the arival of '"' and accumlates all (non '<') characters, till parser again encounters '"' where this time the accumlated string is treaded as attribute value as a result an attribute pair is parsed

The above process is repeated till all the attributes are collected.

16 A.

The DOMBuilder.java extends DefaultHandler.java so all the SAX events are visible to DOMBuilder instance. In my version of the DOMBuilder, DOMBuilder holds an instance of last processed node (currentNode\_)

Every time a start element even is triggered the new element is processed (i.e., the corresponding attributes are add to the new element) the new element is made a child of the currentNode\_ and the new element is made as currentNode\_ (as it is the last processed node) and at the end element event the currentNode\_ is pointed to the parent of the currentNode\_.

Upon characters event a new Text node is created and added as child of the currentNode\_.

17 A.

The way to identify the empty element is by observing if a '/' character is present before '>' of start tag and if parser finds one like that it triggers a start element event and an end element event without any characters between them.

24 A.

Given question gives 6 ways to identify the ignorable-whitespace characters.

1. immediately following some element’s start tag and immediately before another element’s start tag.

By skipping the ignorable-whitespace characters after start tag (skipConsecutiveWhitespaces()) before the while loop in readElement() method.

2. immediately following some element’s start tag and immediately before that element’s end tag.

Means the entire chartext is ignorable-whitespace it is handled by skipping over the ignorable-whitespace characters in char-text (this is done by 2 methods removeIgnorableWhiteSpaceCharactersAtEnd() and removeIgnorableWhiteSpaceCharactersAtStart()).

3. immediately following some element’s start tag and immediately before some nonwhitespace character data.

This is handled by skipping over ignorable-whitespace characters at the begining of the char-text. By calling removeIgnorableWhiteSpaceCharactersAtStart() method.

4. immediately following some non-whitespace data and immediately before some element’s end tag.

This is handled by skipping over ignorable-whitespace characters at the end of the char-text. By calling removeIgnorableWhiteSpaceCharactersAtEnd() method.

5. immediately following some element’s end tag and immediately before another element’s start tag.

This is handled by calling skipConsecutiveWhitespaces(); after reading the element (child) readElement() in (parent) element readElement() method.

6. immediately following some element’s end tag and immediately before another element’s end tag.

This is handled by calling skipConsecutiveWhitespaces(); after reading the element (child) readElement() in (parent) element readElement() method.