Programmieren II

XML/HTML/SGML

Alexander Fraser

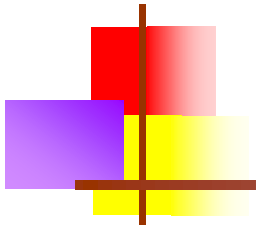
fraser@cl.uni-heidelberg.de

(slides based on material from David Matuszek, U Penn and Terran Lane, UNM)

July 17th, 2014

Introduction

- CL data sets are often stored in XML, HTML or SGML
- At the highest level of abstraction:
  - HTML is a language for formatting web pages
  - XML is a family of languages for storing data
  - SGML is also a family of languages for storing data
    - XML is a subset of SGML
    - SGML is old
    - SGML is still used a lot in publishing even now (for instance, my favorite corpus, the Proceedings of the European Parliament (Europarl))
  - The focus today is XML, I will mention HTML too
    - If you do get something in SGML, look at HTML parsing tools, they often actually support SGML
  - I'm also going to mention JSON, YAML, DOC and PDF
  - And serialization!



HTML and XML, I

XML stands for eXtensible Markup Language

HTML is used to mark up text so it can be displayed to users

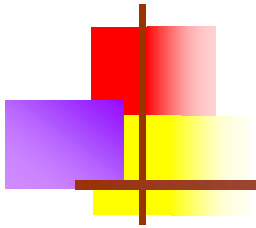
XML is used to mark up data so it can be processed by computers

HTML describes both structure (e.g. <p>, <h2>,) and appearance (e.g.
, <font>, <i>)

XML describes only content, or "meaning"

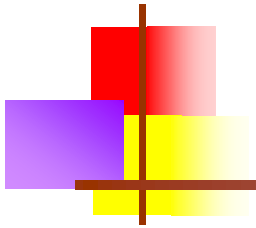
HTML uses a fixed, unchangeable set of tags

In XML, you make up your own tags



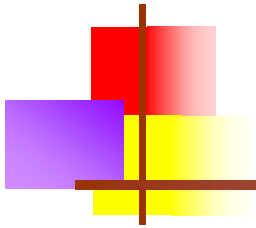
HTML and XML, II

- HTML and XML look similar, because they are both SGML languages (SGML = Standard Generalized Markup Language)
  - Both HTML and XML use elements enclosed in tags (e.g. <body>This is an element</body>)
  - Both use tag attributes (e.g., <font face="Verdana" size="+1" color="red">)
  - Both use entities (<, >, &, ", ')
- More precisely,
  - HTML is defined in SGML
  - XML is a (very small) subset of SGML



HTML and XML, III

- HTML is for humans
  - HTML describes web pages
  - You don’t want to see error messages about the web pages you visit
  - Browsers ignore and/or correct as many HTML errors as they can, so HTML is often sloppy
- XML is for computers
  - XML describes data
  - The rules are strict and errors are not allowed
    - In this way, XML is like a programming language
  - Current versions of most browsers can display XML
    - However, browser support of XML is spotty at best



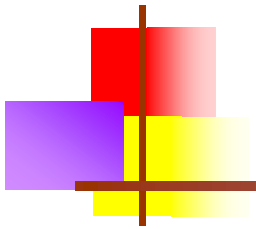
XML-related technologies

- DTD (Document Type Definition) and XML Schemas are used to define legal XML tags and their attributes for particular purposes
- CSS (Cascading Style Sheets) describe how to display HTML or XML in a browser
- XSLT (eXtensible Stylesheet Language Transformations) and XPath are used to translate from one form of XML to another
- DOM (Document Object Model), SAX (Simple API for XML, and JAXP (Java API for XML Processing) are all APIs for XML parsing

Example XML document

```
<?xml version="1.0"?>
<weatherReport>
   <date>7/14/97</date>
   <city>North Place</city>, <state>NX</state>
   <country>USA</country>
   High Temp: <high scale="F">103</high>
   Low Temp: <low scale="F">70</low>
   Morning: <morning>Partly cloudy, Hazy</morning>
   Afternoon: <afternoon>Sunny &amp;amp; hot</afternoon>
   Evening: <evening>Clear and Cooler</evening>
</weatherReport>
```

From: **XML: A Primer**, by Simon St. Laurent



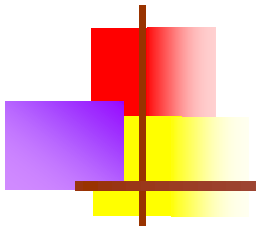
Overall structure

- An XML document may start with one or more processing instructions (PIs) or directives:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="ss.css"?>
```

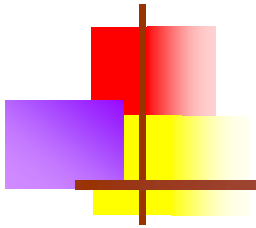
- Following the directives, there must be exactly *one* tag, called the root element, containing all the rest of the XML:

```
<weatherReport>
    …
</weatherReport>
```



XML building blocks

- Aside from the directives, an XML document is built from:
  - elements: high in <high scale="F">103</high>
  - tags, in pairs: <high scale="F">103</high>
  - attributes: <high scale="F">103</high>
  - entities: <afternoon>Sunny & hot</afternoon>
  - character data, which may be:
    - parsed (processed as XML)--this is the default
    - unparsed (all characters stand for themselves)



Elements and attributes

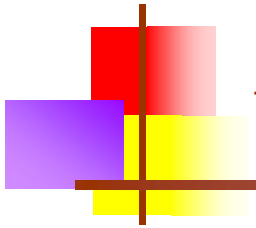
- Attributes and elements are somewhat interchangeable
- Example using just elements:

```
<name>
   <first>David</first>
   <last>Matuszek</last>
</name>
```

- Example using attributes:

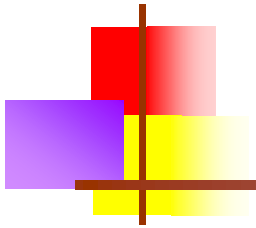
```
<name first="David" last="Matuszek"></name>
```

- You will find that elements are easier to use in your programs--this is a good reason to prefer them
- Attributes often contain metadata, such as unique IDs
- Generally speaking, browsers display only elements (values enclosed by tags), not tags and attributes



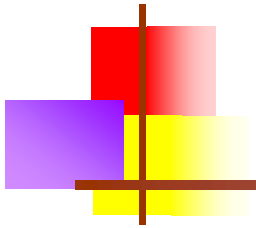
Well-formed XML

- Every element must have *both* a start tag and an end tag, e.g. <name> … </name>
  - But empty elements can be abbreviated: <break />.
  - XML tags are case sensitive
  - XML tags may not begin with the letters xml, in any combination of cases
- Elements must be properly nested, e.g. *not* bold and italic
- Every XML document must have one and only one root element
- The values of attributes must be enclosed in single or double quotes, e.g. <time unit="days">
- Character data cannot contain < or &



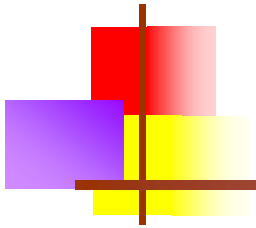
Entities

- Five special characters must be written as entities:
  - `&` for `&` (almost always necessary)
  - `<` for `<` (almost always necessary)
  - `>` for `>` (not usually necessary)
  - `&quot;` for `"` (necessary inside double quotes)
  - `&apos;` for `'` (necessary inside single quotes)
- These entities can be used even in places where they are not absolutely required
- These are the *only* predefined entities in XML



Comments

- `<!-- This is a comment in both HTML and XML -->`
- Comments can be put anywhere in an XML document
- Comments are useful for:
  - Explaining the structure of an XML document
  - Commenting out parts of the XML during development and testing
- Comments are not elements and do not have an end tag
- The blanks after `<!--` and before `-->` are optional
- The character sequence `--` cannot occur in the comment
- The closing bracket *must* be `-->`
- Comments are not displayed by browsers, but can be seen by anyone who looks at the source code



CDATA

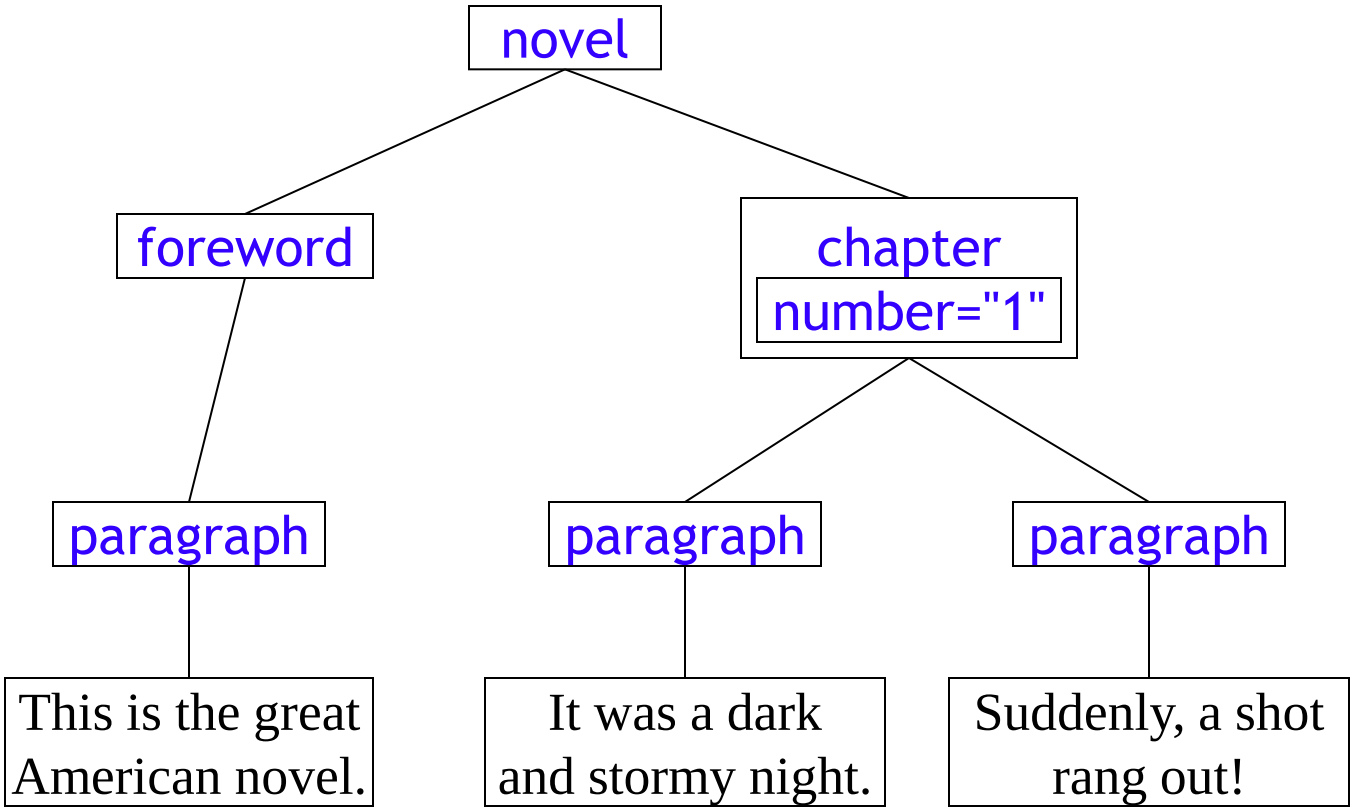
- By default, all text inside an XML document is parsed
- You can force text to be treated as unparsed *character data* by enclosing it in `<![CDATA[ ... ]]>`
- Any characters, even `&` and `<`, can occur inside a CDATA
- Whitespace inside a CDATA is (usually) preserved
- The only real restriction is that the character sequence `]]>` cannot occur inside a CDATA
- CDATA is useful when your text has a lot of illegal characters (for example, if your XML document contains some HTML text)

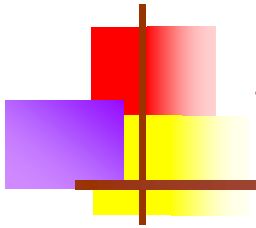
Another well-structured example

```
<novel>
   <foreword>
      <paragraph> This is the great American novel.
      </paragraph>
</foreword>
   <chapter number="1">
      <paragraph>It was a dark and stormy night.
     </paragraph>
      <paragraph>Suddenly, a shot rang out!
      </paragraph>
   </chapter>
</novel>
```

XML as a tree

- An XML document represents a hierarchy; a hierarchy is a tree



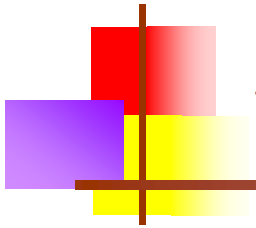


Valid XML

- You can make up your own XML tags and attributes, *but*...
  - ...any program that *uses* the XML must know what to expect!
- A DTD (Document Type Definition) defines what tags are legal and where they can occur in the XML
- An XML document *does not require* a DTD
- XML is well-structured if it follows the rules given earlier
- In addition, XML is valid if it declares a DTD and conforms to that DTD
- A DTD can be included in the XML, but is typically a separate document
- Errors in XML documents will *stop* XML programs
- Some alternatives to DTDs are XML Schemas and RELAX NG

Example XML document, revised

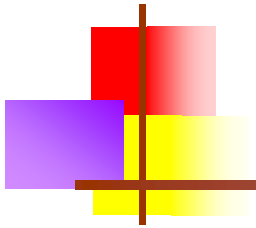
```
<?xml version="1.0"?>
<weatherReport>
  <date>7/14/97</date>
  <place><city>North Place</city>
        <state>NX</state>
        <country>USA</country>
  </place>
  <temperatures><high scale="F">103</high>
                <low scale="F">70</low>
  </temperatures>
  <forecast><time>Morning</time>
            <predict>Partly cloudy, Hazy</predict>
  </forecast>
  <forecast><time>Afternoon</time>
            <predict>Sunny &amp;amp; hot</predict>
  </forecast>
  <forecast><time>Evening</time>
            <predict>Clear and Cooler</predict>
</weatherReport>
```



Viewing XML

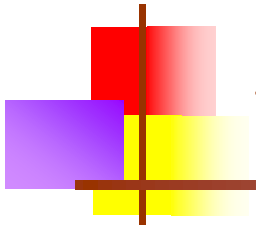
- XML is designed to be processed by computer programs, not to be displayed to humans
- Nevertheless, almost all current browsers can display XML documents
  - They don't all display it the same way
  - They may not display it at all if it has errors
  - For best results, update your browsers to the newest available versions
- Remember:

  HTML is designed to be *viewed*,
  XML is designed to be *used*



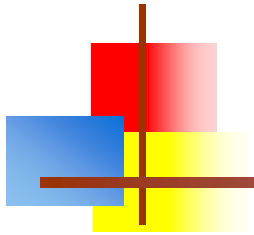
Extended document standards

- You can define your own XML tag sets, but here are some already available:
  - XHTML: HTML redefined in XML
  - SMIL: Synchronized Multimedia Integration Language
  - MathML: Mathematical Markup Language
  - SVG: Scalable Vector Graphics
  - DrawML: Drawing MetaLanguage
  - ICE: Information and Content Exchange
  - ebXML: Electronic Business with XML
  - cxml: Commerce XML
  - CBL: Common Business Library



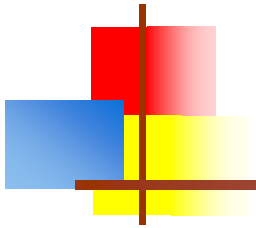
Vocabulary

- SGML: Standard Generalized Markup Language
- XML : Extensible Markup Language
- DTD: Document Type Definition
- element: a start and end tag, along with their contents
- attribute: a value given in the start tag of an element
- entity: a representation of a particular character or string
- PI: a Processing Instruction, to possibly be used by a program that processes this XML
- namespace: a unique string that references a DTD
- well-formed XML: XML that follows the basic syntax rules
- valid XML: well-formed XML that conforms to a DTD



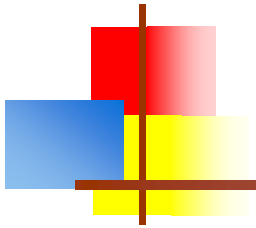
SAX

(Abbreviated)



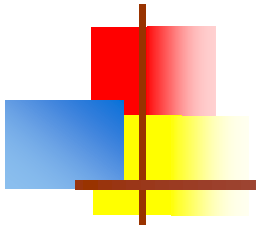
XML Parsers

- SAX and DOM are standards for XML parsers--program APIs to read and interpret XML files
  - DOM is a W3C standard
  - SAX is an ad-hoc (but very popular) standard
    - SAX was developed by David Megginson and is open source
  - XOM is a parser by Elliott Rusty Harold
  - StAX is a newer parser from Sun and BEA Systems
  - Some others are XNI and JDOM
- Unlike many XML technologies, XML parsers are relatively easy



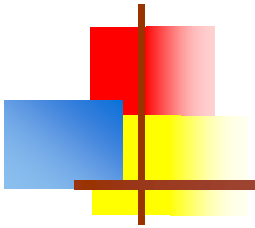
Types of XML parsers

- XML parsers can be classified as *tree-based* or *event-based*
  - Tree-based parsers read the entire XML document into memory and stores it as a tree data structure
    - Tree-based parsers allow random access to the XML, hence are usually more convenient to work with
    - It's usually possible to manipulate the tree and write out the modified XML
    - Parse trees can take up a lot of memory
    - DOM and XOM are tree-based
  - Event-based (or streaming) parsers read sequentially through the XML file
    - Event-based parsers are faster and take very little memory—this is important for large documents and for web sites
    - SAX and StAX are event-based



Types of streaming XML parsers

- Streaming XML parsers can be classified as *push* or *pull* parsers
  - Push parsers take control and call your methods with the XML constituents as they are encountered
    - This type of programming is unnatural for Java programmers
    - SAX and XNI are push parsers
  - Pull parsers let you ask for the next XML constituent
    - Pull parsers are similar to iterators
    - StAX is a pull parser

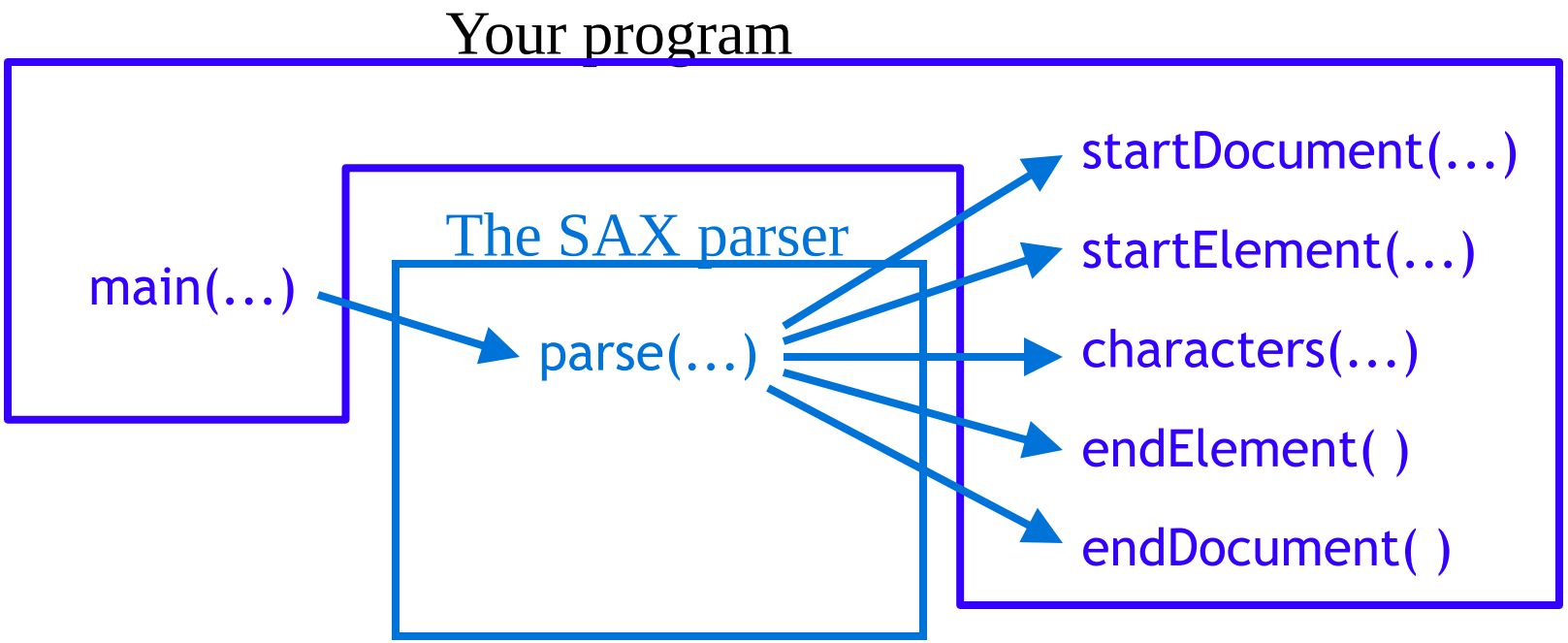


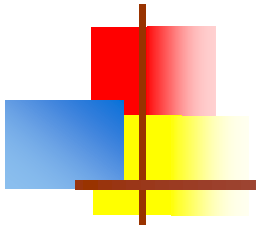
SAX vs. StAX

- At this point, there seems to be no reason to use SAX rather than StAX--*if* you have a choice
  - SAX and StAX are both streaming parsers
  - StAX is faster and simpler
  - With StAX your program has control, rather than being controlled by the parser
    - This means:
      - You can choose what tags to look at, rather than having to deal with them all
      - You can stop whenever you like
- However,
  - StAX is newer, so many existing projects use SAX
  - Many ideas, such as the use of factories, are the same for each

SAX uses callbacks

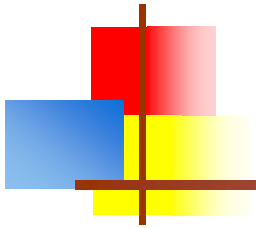
- SAX works through callbacks: you call the parser, it calls methods that you supply





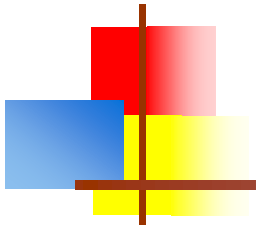
Simple SAX program

- The following program is adapted from CodeNotes® for XML by Gregory Brill, pages 158-159
- The program consists of two classes:
  - Sample -- This class contains the main method; it
    - Gets a factory to make parsers
    - Gets a parser from the factory
    - Creates a Handler object to handle callbacks from the parser
    - Tells the parser which handler to send its callbacks to
    - Reads and parses the input XML file
  - Handler -- This class contains handlers for three kinds of callbacks:
    - startElement callbacks, generated when a start tag is seen
    - endElement callbacks, generated when an end tag is seen
    - characters callbacks, generated for the contents of an element



The Sample class, I

```
import javax.xml.parsers.*; // for both SAX and DOM
import org.xml.sax.*;
import org.xml.sax.helpers.*;


// For simplicity, we let the operating system handle exceptions
// In "real life" this is poor programming practice
public class Sample {
    public static void main(String args[]) throws Exception {
        // Create a parser factory
        SAXParserFactory factory = SAXParserFactory.newInstance();
        // Tell factory that the parser must understand namespaces
        factory.setNamespaceAware(true);
        // Make the parser
         SAXParser saxParser = factory.newSAXParser();
         XMLReader parser = saxParser.getXMLReader();
```

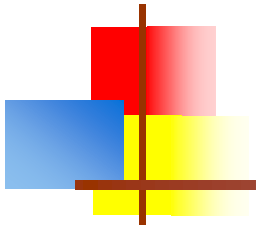


The Sample class, II

- In the previous slide we made a parser, of type XMLReader

```
        // Create a handler (Handler is my class)
        Handler handler = new Handler();
        // Tell the parser to use this handler
        parser.setContentHandler(handler);
        // Finally, read and parse the document
        parser.parse("hello.xml");
    } // end of Sample class
```

- You will need to put the file hello.xml :
  - In the same directory, if you run the program from the command line
  - Or where it can be found by the particular IDE you are using



The Handler class, I

- public class Handler extends DefaultHandler {
  - DefaultHandler is an adapter class that defines these methods and others as do-nothing methods, to be overridden as desired
  - We will define three very similar methods to handle (1) start tags, (2) contents, and (3) end tags--our methods will just print a line
  - Each of these three methods could throw a SAXException

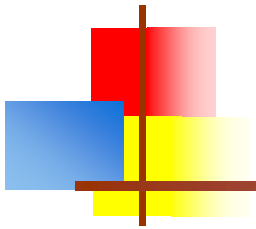
```
// SAX calls this method when it encounters a start tag
public void startElement(String namespaceURI,
                         String localName,
                         String qualifiedName,
                         Attributes attributes)
        throws SAXException {
    System.out.println("startElement: " + qualifiedName);
}
```

The Handler class, II

```
// SAX calls this method to pass in character data
public void characters(char ch[ ], int start, int length)
        throws SAXException {
    System.out.println("characters: \"" +
                        new String(ch, start, length) + "\"");
}
```

```
// SAX call this method when it encounters an end tag
public void endElement(String namespaceURI,
                       String localName,
                       String qualifiedName)
        throws SAXException {
    System.out.println("Element: /" + qualifiedName);
}
```

```
} // End of Handler class
```



Results

- If the file hello.xml contains:

```
<?xml version="1.0"?>
<display>Hello World!</display>
```

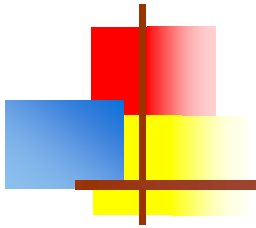
- Then the output from running java Sample will be:

```
startElement: display
characters: "Hello World!"
Element: /display
```

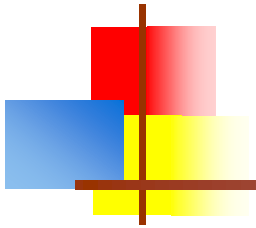
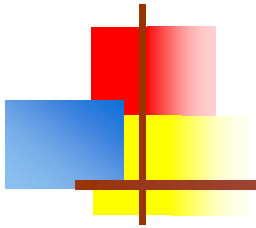
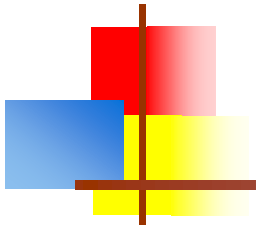
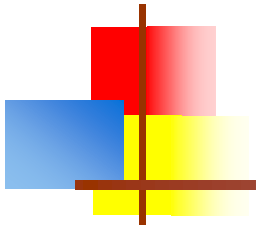
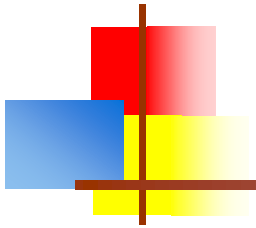
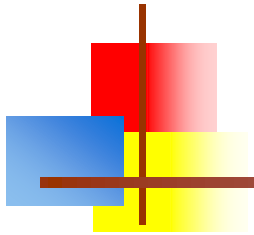
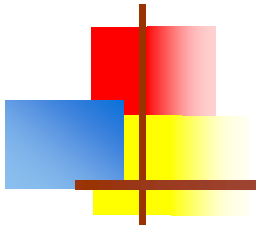
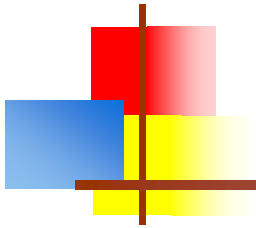
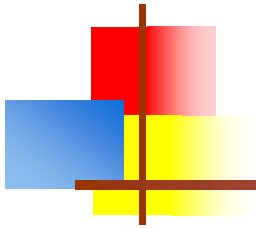
More results

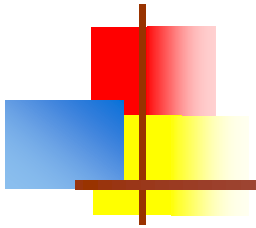
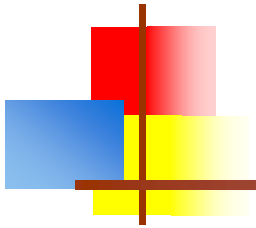
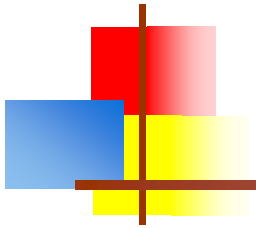
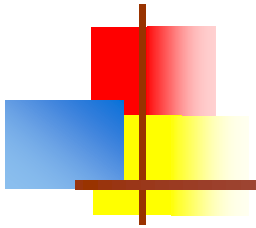
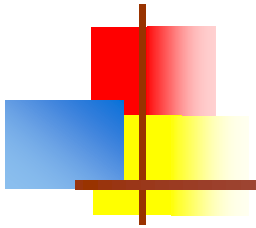
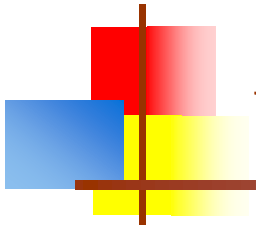
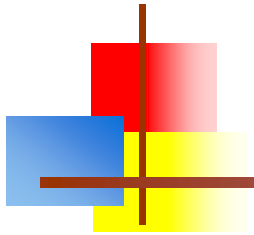
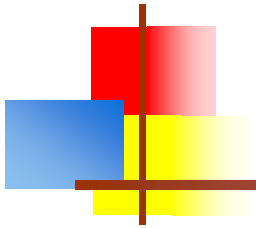
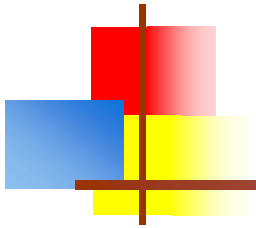
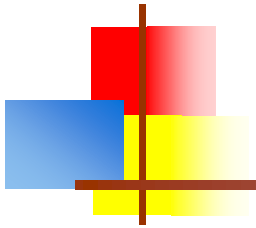
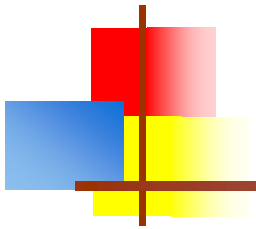
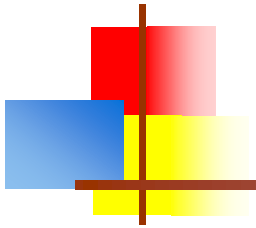
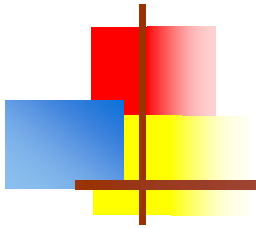
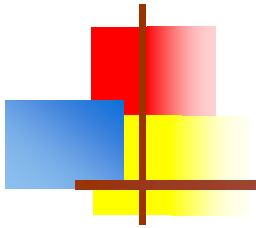
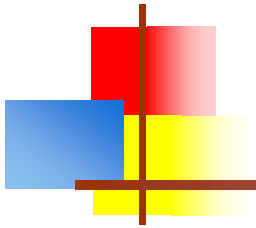
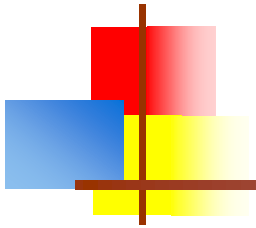
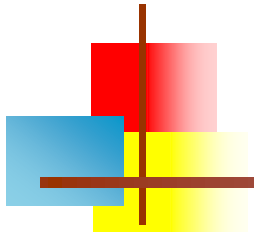
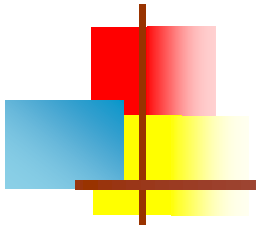
- Now suppose the file hello.xml contains:
  - <?xml version="1.0"?>
    <display>
      Hello World!
    </display>
- Notice that the root element, <display>, now contains a nested element and some whitespace (including newlines)
- The result will be as shown at the right:

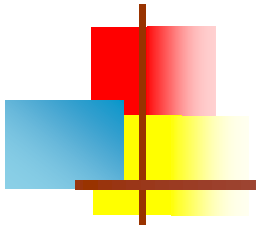
```
startElement: display
characters: "" // empty string
characters: "
"    // newline
characters: "      " // spaces
startElement: i
characters: "Hello"
endElement: /i
characters:  "World!"
characters: "
"    // another newline
endElement: /display
```



Factories

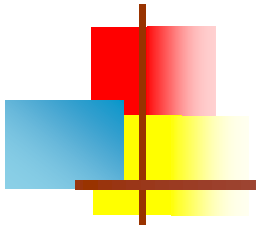
- SAX uses a parser factory
- A factory is an alternative to constructors
- Factories allow the programmer to:
  - Decide whether or not to create a new object
  - Decide what kind (subclass, implementation) of object to create
- Trivial example:
  - ```
    class TrustMe {
        private TrustMe() { } // private constructor

        public static TrustMe makeTrust() { // factory method
            if ( /* test of some sort */)
                return new TrustMe();
            }
        }
    }
    ```
    ```



Parser factories

- To create a SAX parser factory, call this method: SAXParserFactory.newInstance()
  - This returns an object of type SAXParserFactory
  - It may throw a FactoryConfigurationError
- You can then customize your parser:
  - public void setNamespaceAware(boolean awareness)
    - Call this with true if you are using namespaces
    - The default (if you don't call this method) is false
  - public void setValidating(boolean validating)
    - Call this with true if you want to validate against a DTD
    - The default (if you don't call this method) is false
    - Validation will give an *error* if you *don't* have a DTD



Getting a parser

- Once you have a SAXParserFactory set up (say it's named factory), you can create a parser with:

```
SAXParser saxParser = factory.newSAXParser();
XMLReader parser = saxParser.getXMLReader();
```

- Note: older texts may use Parser in place of XMLReader
  - Parser is SAX1, not SAX2, and is now deprecated
  - SAX2 supports namespaces and some new parser properties
- Note: SAXParser is not thread-safe; to use it in multiple threads, create a separate SAXParser for each thread
  - This is unlikely to be a problem in small projects



Declaring which handler to use

- Since the SAX parser will be calling our methods, we need to supply these methods
- In the example these are in a separate class, Handler
- We need to tell the parser where to find the methods:

```
Handler handler = new Handler();
parser.setContentHandler(handler);
```

- These statements could be combined:

```
parser.setContentHandler(new Handler());
```

- Finally, we call the parser and tell it what file to parse:

```
parser.parse("hello.xml");
```

- Everything else will be done in the handler methods



SAX handlers

- A callback handler for SAX must implement these four interfaces:
  - interface ContentHandler
    - This is the most important interface
    - Handles elements (tags), attributes, and text content
    - Again, done by callbacks--you have to supply a method for each type
  - interface DTDHandler
    - Handles *only* notation and unparsed entity declarations
  - interface EntityResolver
    - Does customized handling for external entities
  - interface ErrorHandler
    - Must be implemented or parsing errors will be *ignored!*
- Adapter classes are provided for these interfaces



Whitespace

- Whitespace is usually a problem when parsing XML
- A *nonvalidating* parser cannot ignore whitespace, because it cannot distinguish it from real data
  - You have to write code to recognize and discard whitespace
- A *validating* parser ignores whitespace where character data is not allowed
  - For processing XML, this is usually what you want
  - However, if you are manipulating and *writing out* XML, discarding whitespace ruins your indentation
  - To capture ignorable whitespace, SAX provides a method ignorableWhitespace that you can override



DOM



Difference between SAX and DOM

- DOM reads the entire XML document into memory and stores it as a tree data structure
- SAX reads the XML document and sends an event for each element that it encounters
- Consequences:
  - DOM provides "random access" into the XML document
  - SAX provides *only* sequential access to the XML document
  - DOM is slow and requires huge amounts of memory, so it cannot be used for large XML documents
  - SAX is fast and requires very little memory, so it can be used for huge documents (or large numbers of documents)
    - This makes SAX much more popular for web sites
  - Some DOM implementations have methods for *changing* the XML document in memory; SAX implementations do not



Simple DOM program, I

- This program is adapted from CodeNotes® for XML by Gregory Brill, page 128
- ```
  import javax.xml.parsers.*;
  import org.w3c.dom.*;
  ```
- ```
  public class SecondDom {
      public static void main(String args[]) {
          try {
              ...Main part of program goes here...
          } catch (Exception e) {
              e.printStackTrace(System.out);
          }
      }
  }
  ```



Simple DOM program, II

- First we need to create a DOM parser, called a "DocumentBuilder"
- The parser is created, *not* by a constructor, but by calling a static factory method
  - This is a common technique in advanced Java programming
  - The use of a factory method makes it easier if you later switch to a different parser

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder =
    factory.newDocumentBuilder();
```



Simple DOM program, III

- The next step is to load in the XML file
- Here is the XML file, named hello.xml:

```
<?xml version="1.0"?>
<display>Hello World!</display>
```

- To read this file in, we add the following line to our program:

```
Document document = builder.parse("hello.xml");
```

- Notes:
  - document contains the entire XML file (as a tree); it *is* the Document Object Model
  - If you run this from the command line, your XML file should be in the same directory as your program
  - An IDE may look in a different directory for your file; if you get a java.io.FileNotFoundException, this is probably why



Simple DOM program, IV

- The following code finds the content of the root element and prints it:

```
Element root = document.getDocumentElement();
Node textNode = root.getFirstChild();
System.out.println(textNode.getNodeValue());
```

- The output of the program is: Hello World!



Reading in the tree

- The parse method reads in the entire XML document and represents it as a tree in memory
  - For a large document, parsing could take a while
  - If you want to interact with your program while it is parsing, you need to parse in a separate thread
    - Once parsing starts, you cannot interrupt or stop it
    - Do not try to access the parse tree until parsing is done
- An XML parse tree may require up to ten times as much memory as the original XML document
  - If you have a lot of tree manipulation to do, DOM is much more convenient than SAX
  - If you *don't* have a lot of tree manipulation to do, consider using SAX instead



Structure of the DOM tree

- The DOM tree is composed of Node objects
- Node is an interface
  - Some of the more important subinterfaces are Element, Attr, and Text
    - An Element node may have children
    - Attr and Text nodes are leaves
  - Additional types are Document, ProcessingInstruction, Comment, Entity, CDATASection and several others
- Hence, the DOM tree is composed entirely of Node objects, but the Node objects can be downcast into more specific types as needed



Manipulating DOM trees

- DOM, unlike SAX, gives you the ability to create and modify XML trees
- There are a few roadblocks along the way
  - Practically everything in the JAXP implementation is an interface, with a few abstract classes
  - Interfaces, such as Node, don't have constructors; this makes it hard to get started
  - Since DOM was designed to be applicable from a number of languages, many things are not done "the Java way"
- Once you get past these problems, the individual methods are pretty straightforward
  - Even with straightforward methods, working with trees is seldom simple



Writing out the DOM as XML

- There are no Java-supplied methods for writing out a DOM as XML
- Writing out a DOM is conceptually simple—it's just a tree walk
  - However, there are a *lot* of details—various node types and so on—so doing a good job isn't complicated, but it *is* lengthy

DOM

- I'm skipping a lot of details on DOM
- Consider using DOM only if you want the full tree in memory (this is unusual)



StAX

Streaming API for XML



XML parser comparisons

- DOM is
  - Memory intensive
  - Read-write
  - Typically used for documents smaller than 10 MB
- SAX is
  - Memory efficient
  - Read only
  - Typically used for documents larger than 10 MB
- StAX is
  - Memory efficient
  - Read-write
  - Appropriate for documents of all sizes
  - Easier to use than DOM or SAX
- Source: Murach's Java SE 6



Creating a StAX reader

- Everything you need specifically for working with StAX is in javax.xml.stream
  - import javax.xml.stream.*
- As with SAX and DOM, you first need to get a factory by calling a static method
  - XMLInputFactory factory = XMLInputFactory.newInstance();
- You can set some properties on this factory
  - factory.setProperty("javax.xml.stream.isValidating", "true");
  - factory.setProperty("javax.xml.stream.isCoalescing", "true");
- You will need to have some way to supply the XML, usually from an InputStream or a Reader
  - FileReader fileReader = new FileReader("***somefile***.xml");
  - This line could throw a FileNotFoundException
- You can then create a reader for the XML
  - XMLStreamReader reader = factory.createXMLStreamReader(fileReader);
  - This line could throw an XMLStreamException



Using the StAX parser

- A StaAX parser (reader) behaves similar to an Iterator
  - The boolean hasNext() method tells you if there is another "event" to read
  - The int next() method reads the next event
    - However, what it returns is an **int** that tells what kind of event it was
    - Some values are START_ELEMENT, END_ELEMENT, ATTRIBUTE, CHARACTERS (content), COMMENT, SPACE, END
- After doing next(), the parser has a "current element," and can be asked questions about that element
  - For example, getLocalName() returns the name of the current element
- It is important to remember that the parser only moves forward; it's easy to go past the information you need
- It's easier to parse a document if you know its structure
  - A validating parser can check the structure against a DTD
  - isValidating *is* a supported feature of Java's current StAX parser



Using getLocalName()

- getLocalName() returns the tag name as a String
- Your code will probably look something like this:
  - String name = reader.getLocalName();
    if (name.equals(***someTag***)) { … }
    else if (name.equals(***someOtherTag***)) { … }
    else if (name.equals(***someOtherTag***)) { … }
    …
- Or you could use a switch statement (Java 1.7+)



Constants

- next() moves to the next event, and returns an int to tell what type it is
  - START_DOCUMENT
  - END_DOCUMENT
  - START_ELEMENT
  - END_ELEMENT
  - ATTRIBUTE
  - CHARACTERS
  - COMMENT
  - SPACE
  - DTD
  - PROCESSING_INSTRUCTION
  - NAMESPACE
  - CDATA
  - ENTITY_REFERENCE
- These are all constants defined in the XMLStreamReader object



Methods

- There are numerous methods defined in the XMLStreamReader object; here are some of the most important:
  - boolean hasNext() – true if there is another "event"
  - int next() – gets next parsing event, return its type
  - int nextTag() – gets next start or end element, return its type
  - getEventType() – gets next parsing event, returns its type
  - getLocalName() – gets the name of the current element or entity reference
  - getAttributeCount() – gets the number of attributes of the current element
  - getAttributeLocalName(***index***) – gets the name of the current attribute
  - getAttributeValue(***index***) – gets the value of the current attribute
  - getElementText() – returns the coalesced text of a START_ELEMENT; after the call, the current event will be the corresponding END_ELEMENT
  - getText() – returns the text value of a CHARACTERS, COMMENT, ENTITY_REFERENCE, CDATA, SPACE, or DTD



Creating a StAX writer

- Everything you need specifically for working with StAX is in javax.xml.stream
  - import javax.xml.stream.*
- As with SAX and DOM, you first need to get a factory by calling a static method
  - XMLOutputFactory factory = XMLOutputFactory.newInstance();
- You will need to have somewhere to put the XML, usually to an OutputStream or a Writer
  - FileWriter fileWriter = new FileWriter("***somefile***.xml");
  - This line could throw an IOException
- You can then create a writer for the XML
  - XMLStreamWriter writer = factory.createXMLStreamWriter(fileWriter);
  - This line could throw an XMLStreamException



Methods

- There are numerous methods defined in the XMLStreamWriter object; here are some of the most important:
  - writeStartDocument(*version*) – writes the XML header
  - writeStartElement(*name*) – writes a start tag
  - writeAttribute(*name*, *value*) – writes an attribute
  - writeCharacters(*value*) – writes text, encoding < > & characters
  - writeComment(*value*) – writes a comment
  - writeDTD(*value*) – writes an (entire) DTD
  - writeEndElement() – writes an end tag for the current start tag
  - flush() – forces buffered output to be actually written
  - close() – close the writer



Conclusions

- Advantages of StAX:
  - Simpler to use than SAX, in part because it doesn't use callbacks
  - Can write XML files as well as read them
- Disadvantages of StAX
  - Practically forces you to use an extended if-then-else to recognize what has been parsed
  - As with SAX, you can only go forward
- Comparison with DOM:
  - StAX is much faster, more memory efficient, and somewhat simpler
  - DOM lets you manipulate trees in memory

HTML

- HTML/SGML parsers have to be much more robust to incorrect markup than XML parsers
- I would recommend JSoup
- JSoup example here:
  - http://www.programcreek.com/2012/05/parse-html-in-java/
  - Downloads a web page and extracts links
- TagSoup may also be of interest

DOC/PDF

- If you need to process DOC (or DOCX, or other Microsoft formats), use:
  - http::://poi.apache.org
- If you need to process PDF:
  - First test the PDF file to see if you can copy/paste the text.
  - If you can copy/paste, then you can use:
  - http://pdfbox.apache.org



JSON



JSON example

- "JSON" stands for "JavaScript Object Notation"
  - Despite the name, JSON is a (mostly) language-independent way of specifying objects as name-value pairs
- Example (http://secretgeek.net/json_3mins.asp):
  - ```
    {"skillz": {
        "web":[
            { "name": "html",
              "years": 5
            },
            { "name": "css",
              "years": 3
            }]
        "database":[
            { "name": "sql",
              "years": 7
            }]
    }}
    ```
    ```



JSON syntax, I

- An *object* is an unordered set of name/value pairs
  - The pairs are enclosed within braces, { }
  - There is a colon between the name and the value
  - Pairs are separated by commas
  - Example: { "name": "html", "years": 5 }
- An *array* is an ordered collection of values
  - The values are enclosed within brackets, [ ]
  - Values are separated by commas
  - Example: ["html", ”xml", "css"]

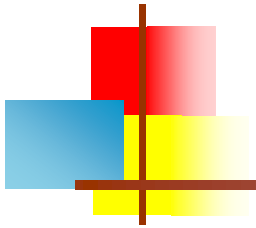


JSON syntax, II

- A *value* can be: A string, a number, true, false, null, an object, or an array
  - Values can be nested
- *Strings* are enclosed in double quotes, and can contain the usual assortment of escaped characters
- *Numbers* have the usual C/C++/Java syntax, including exponential (E) notation
  - All numbers are decimal--no octal or hexadecimal
- *Whitespace* can be used between any pair of tokens

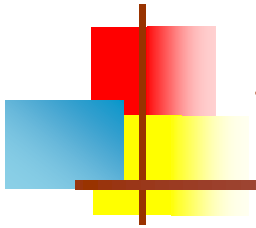
JSON in Java

- The Jackson JSON library is recommended
- It can write Java Objects into their JSON representation
- It can also read JSON and instantiate a Java Object



Comparison of JSON and XML

- Similarities:
  - Both are human readable
  - Both have very simple syntax
  - Both are hierarchical
  - Both are language independent
  - Both can be used by Ajax
- Differences:
  - Syntax is different
  - JSON is less verbose
  - JSON includes arrays
  - XML can be validated



YAML

- YAML can be taken as an acronym for either
  - Yet Another Markup Language
  - YAML Ain't Markup Language
- Like JSON, the purpose of YAML is to represent typical data types in human-readable notation
- YAML is (almost) a superset of JSON, with many more capabilities (lists, casting, etc.)
  - Except: YAML doesn't handle escaped Unicode characters
  - Consequently, JSON can be parsed by YAML parsers
- When JSON isn't enough, consider YAML

YAML Example

- Cool example: http://en.wikipedia.org/wiki/YAML#Sample_document

Serialization

- Serialization allows us to save and load Java objects
- (Based on material from Terran Lane, UNM)

So you want to save your data…

- Common problem:
  - You've built a large, complex object
    - Spam/Normal statistics tables
    - Game state
    - Database of student records
    - Etc…
  - Want to store on disk and retrieve later
  - Or: want to send over network to another Java process
- In general: want your objects to be *persistent* -- outlive the current Java process

Answer I: Homebrew file formats

- You've got file I/O nailed, so…
- Write a set of methods for saving/loading each class that you care about

```
public class MyClass {
  public void saveYourself(Writer o)
    throws IOException { … }
  public static MyClass loadYourself(Reader r)
    throws IOException { … }
}
```

Coolnesses of Approach 1:

- Can produce arbitrary file formats
- Know exactly what you want to store and get back/don't store extraneous stuff
- Can build file formats to interface w/ other codes/programs
  - XML
  - Tab-delimited/spreadsheet
  - Etc.
- If your classes are nicely hierarchical, makes saving/loading simple

Saving/Loading Recursive Data Structs

```
public interface Saveable {
  public void saveYourself(Writer w)
    throws IOException;
  // should also have this
  // public static Object loadYourself(Reader r)
  //   throws IOException;
  // but you can't put a static method in an
  // interface in Java
}
```

Saving, cont'd

```
public class MyClassA implements Saveable {
  public MyClassA(int arg) {
    // initialize private data members of A
  }
  public void saveYourself(Writer w)
    throws IOException {
    // write MyClassA identifier and private data on
    // stream w
  }
  public static MyClassA loadYourself(Reader r)
    throws IOException {
    // parse MyClassA from the data stream r
    MyClassA tmp=new MyClassA(data);
    return tmp;
  }
}
```

Saving, cont'd

```
public class MyClassB implements Saveable {
  public void MyClassB(int arg) { … }
  private MyClassA _stuff;
  public void saveYourself(Writer w) {
    // write ID for MyClassB
    _stuff.saveYourself(w);
    // write other private data for MyClassB
    w.flush();
  }
  public static MyClassB loadYourself(Reader r) {
    // parse MyClassB ID from r
    MyClassA tmp=MyClassA.loadYourself(r);
    // parse other private data for MyClassB
    return new MyClassB(tmp);
  }
}
```

Painfulnesses of Approach 1:

- This is called *recursive descent parsing* (and formatting)
- We'll use it in project 2, and there are plenty of places in the Real World (TM) where it's terribly useful.
- But... It's also a pain in the a**
- If all you want to do is store/retrieve data, do you *really* need to go to all of that effort?
- Fortunately, no. Java provides a shortcut that takes a lot of the work out.

Approach 2: Enter Serialization...

- Java provides the serialization mechanism for object persistence
- It essentially automates the grunt work for you
- Short form:

```
public class MyClassA implements Serializable { ... }
// in some other code elsewhere...
MyClassA tmp=new MyClassA(arg);
FileOutputStream fos=new FileOutputStream("some.obj");
ObjectOutputStream out=new ObjectOutputStream(fos);
out.writeObject(tmp);
out.flush();
out.close();
```

In a bit more detail...

- To (de-)serialize an object, it must `implements Serializable`
  - All of its data members must also be marked serializable
  - And so on, recursively...
  - Primitive types (`int`, `char`, etc.) are all serizable automatically
  - So are Strings, most classes in java.util, etc.
- This saves/retrieves the entire object graph, including ensuring uniqueness of objects

To read it back in

```
try
{
      fis = new FileInputStream(filename);
      in = new ObjectInputStream(fis);
      myInstance = (MyClassA)in.readObject();
      in.close();
}
catch(IOException ex) {
       ex.printStackTrace();
}
catch(ClassNotFoundException ex) {
       ex.printStackTrace();
}
```

Note that if the class definition is different, this will fail with a class not found exception!

Literature

- Ullenboom, Ch.
  Java ist auch eine Insel (Chapter 18 (especially 18.5) and 17.10)
  Galileo Computing, 2012
- Head First Java
  Chapter 14