Java for Advanced Programmers

Student Workbook

itcourseware
Training Materials For IT Professionals

JAVA FOR ADVANCED PROGRAMMERS

Contributing Authors: John Crabtree, Danielle Hopkins, Julie Johnson, Channing Lovely, Mike Naseef, Jamie Romero, Rob Roselius, Rob Seitz, and Rick Sussenbach.

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editor: Jan Waleri

Editorial Staff: Danielle North

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Jimmy Ball, Larry Burley, Roger Jones, Joe McGlynn, Jim McNally, Mike Naseef, Richard Raab, and Todd Wright, and the many students who have offered comments, suggestions, criticisms, and insights.

CONTENTS

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Access XML content with the Java API for XML Processing (JAXP).
- Store and retrieve a serialized Java object.
- Use buffers and channels from Java's New I/O packages.
- Use reflection classes to examine objects and classes at runtime.
- Create client/server Java applications using Remote Method Invocation (RMI).
- Bind and lookup objects in a naming service using the Java Naming and Directory Interface (JNDI).

COURSE OVERVIEW

- **Audience**: This course is designed for Java programmers who wish to increase their depth of knowledge in Java programming and explore the uses of the various advanced packages.

- **Prerequisites**: *Intermediate Java Programming* or equivalent experience.

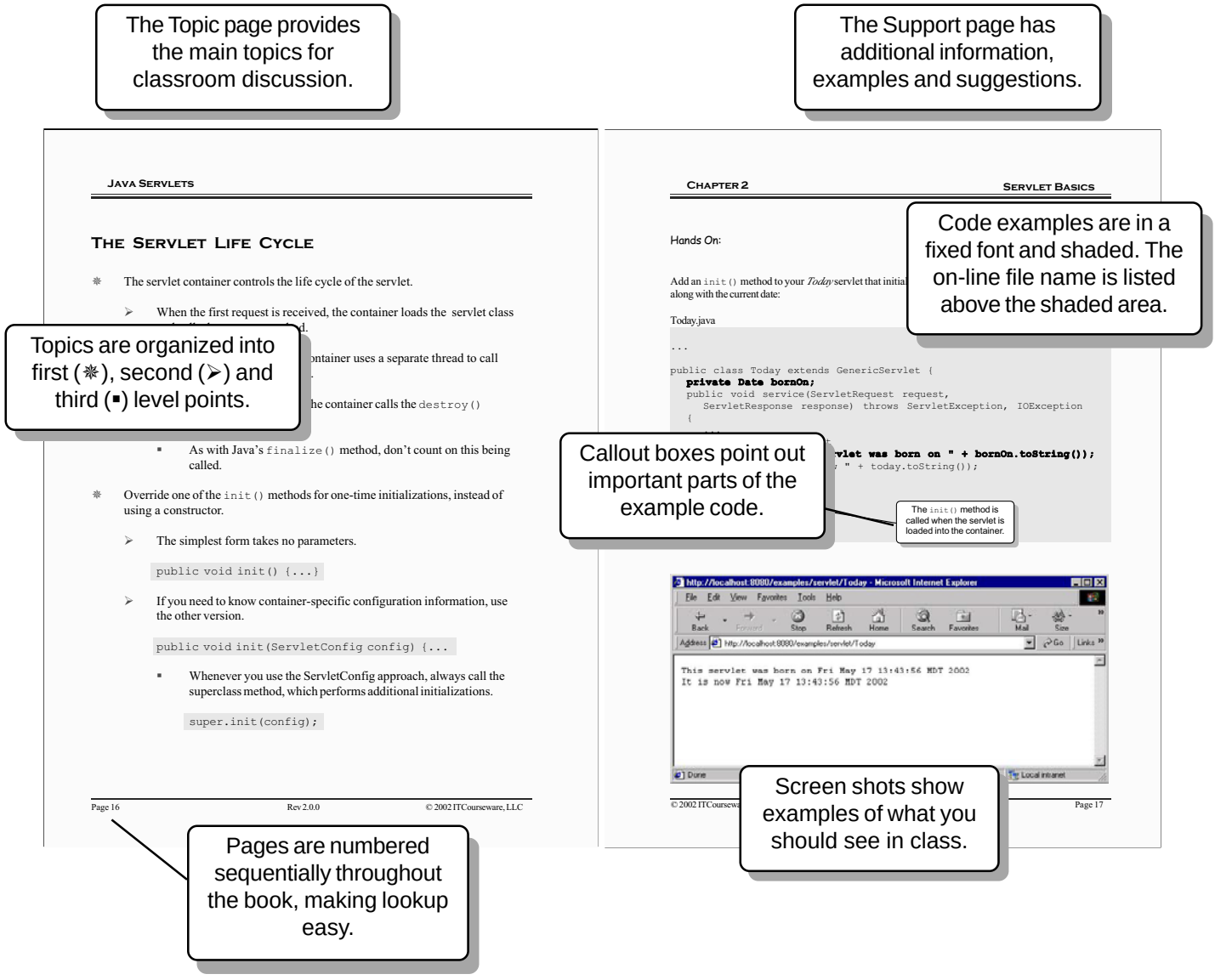
- **Classroom Environment:**

  - One Java development environment per student.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. *Hands On* sections provide opportunities for practical application of key concepts. *Try It* and *Investigate* sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.



SUGGESTED REFERENCES

Arnold, Ken, James Gosling, and David Holmes. 2005. ***The Java Programming Language***. Addison-Wesley, Reading, MA. ISBN 0321349806.

Bloch, Joshua. 2001. ***Effective Java Programming Language Guide***. Addison-Wesley, Reading, MA. ISBN 0201310058.

Eckel, Bruce. 2002. ***Thinking in Java***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0131002872.

Fisher, Maydene, Jonathan Bruce, and Jon Ellis. 2003. ***JDBC API Tutorial and Reference: Universal Data Access for the Java2 Platform, Third Edition***. Addison-Wesley, Reading, MA. ISBN 0321173848.

Flanagan, David. 2005. ***Java in a Nutshell, Fifth Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596007736.

Flanagan, David and Brett McLaughlin. 2004. ***Java 5.0 Tiger: A Developer's Notebook***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596007388.

Freeman, Elizabeth, et al. 2004. ***Head First Design Patterns***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596007124.

Gamma, Erich, et al. 1995. ***Design Patterns***. Addison-Wesley, Reading, MA. ISBN 0201633612.

Gordon, Rob. 1998. ***Essential JNI: Java Native Interface***. Prentice Hall, Upper Saddle River, NJ. ISBN 0136798950.

Haggar, Peter. 2000. ***Practical Java Programming Language Guide***. Addison-Wesley, Reading, MA. ISBN 0201616467.

Harold, Elliotte Rusty. 2004. ***Java Network Programming, Third Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596007213.

Hitchens, Ron. 2002. ***Java NIO***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596002882.

Horstmann, Cay and Gary Cornell. 2004. ***Core Java 2, Volume I: Fundamentals***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0131482025.

Horstmann, Cay S. and Gary Cornell. 2004. ***Core Java 2, Volume II: Advanced Features, Seventh Edition***. Prentice Hall, Upper Saddle River, NJ. ISBN 0131118269.

Oaks, Scott. 2001. ***Java Security, Second Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596001576.

Richardson, W. Clay, Donald Avondolio, Joe Vitale, Scot Schrager, Mark W. Mitchell, and Jeff Scanlon. 2005. ***Professional Java, JDK***. Wrox Press, London, England. ISBN 0764574868.

Sierra, Kathy and Bert Bates. 2005. ***Head First Java***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596009208.

White, Seth, Maydene Fisher, et al. 2003. ***JDBC API Tutorial and Reference***. Addison-Wesley, Reading, MA. ISBN 0321173848.

van der Linden, Peter. 2004. ***Just Java 2***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 013142114

http://www.oracle.com/technetwork/java/index.html
http://www.javaworld.com

CHAPTER 2 - PROCESSING XML WITH JAVA – JAXP

OBJECTIVES

- Describe the Java API for XML Processing (JAXP).
- Use SAX callbacks to parse XML.
- Traverse and manipulate the DOM tree.
- Use JAXP to transform an XML document with XSLT.

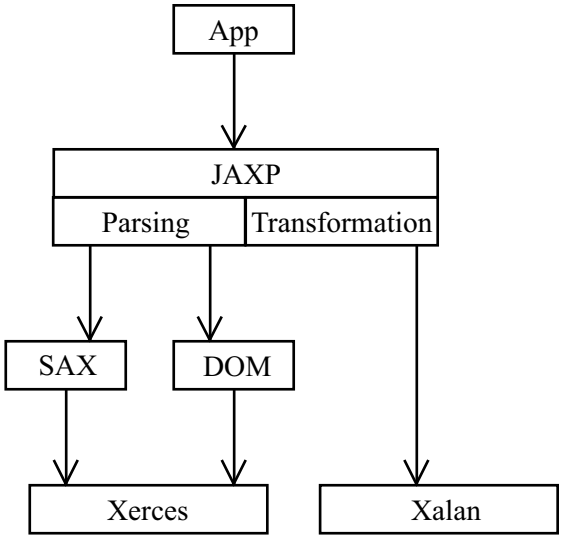
THE JAVA API FOR XML PROCESSING

- Java API for XML Processing (JAXP) enables Java applications to parse and transform XML documents without constraint from a particular XML implementation.
  - You can develop programs independent of the underlying XML parser by using the JAXP APIs; then the XML parser can be changed without changing a single line of application code.
  - The underlying XML parser can be determined at runtime by setting the appropriate switch.

    ```
    java -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser
    ```

- JAXP supports both object-based and event-based parsing of XML files.
  - Object-based parsing using the Document Object Model (DOM) involves the creation of an internal data structure to represent the XML document content.
  - Event-based parsing using Simple API for XML (SAX) treats the contents of the XML document as a series of events and calls various methods in response to those events.
- JAXP also provides a common interface for working with XSLT.
  - XSLT is an XML specification that provides instructions on how to transform a source document into a result document.
  - You can create a new XML, HTML, or text document using an XSLT Processor.
  - Just as SAX and DOM parsers can be switched at runtime, so can XSLT processors.

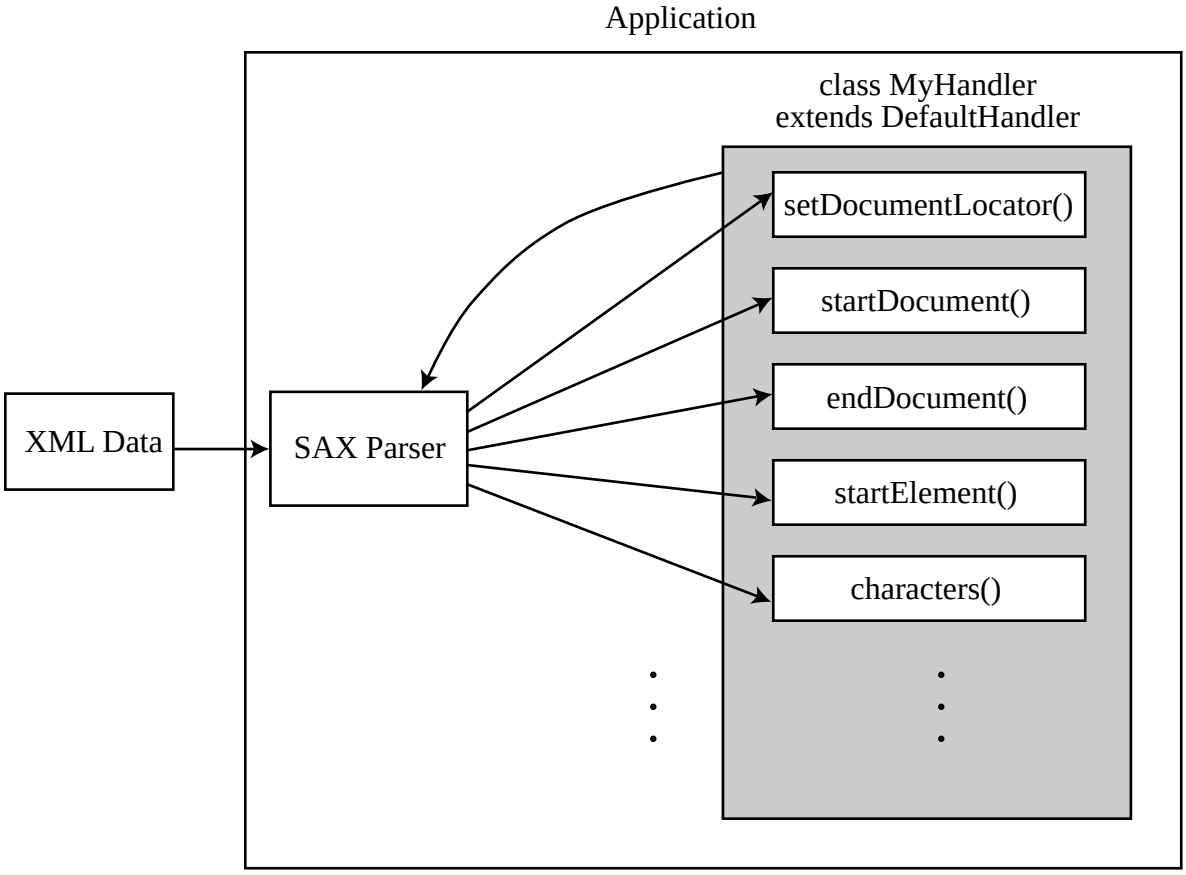
JAXP is bundled with the JDK, standard edition, version 1.4 or later.



Xerces and Xalan are Apache implementations that come bundled with Java 5.

INTRODUCTION TO SAX PARSING

- SAX uses an *event-driven model* to parse XML data.
  - The XML content is seen as a series of events, triggering the SAX parser to call various handler methods.
- SAX has the advantage of low memory consumption.
  - The entire document does not need to be loaded into memory at one time, thus enabling SAX to parse larger documents.
- A disadvantage is the necessity of maintaining event state in your application code.
- A SAX-compliant parser specifies handler interfaces to respond to parser events.
  - The **ContentHandler** interface is used to respond to basic parsing events.
  - The **ErrorHandler** interface methods are called in response to problems in the XML file.
- The **DefaultHandler** class implements all the basic SAX interfaces and provides default implementations.
  - By extending the **DefaultHandler** you can implement only the methods you are interested in.
  - As the parser operates on the XML data, it calls your methods in response to the document.


Application
class MyHandler
extends DefaultHandler
setDocumentLocator()
startDocument()
endDocument()
startElement()
characters()
XML Data
SAX Parser

SAXPARSER AND JAXP

- There are four basic steps when using JAXP with SAX:

1. Get an instance of the **SAXParserFactory**.

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

2. Use the **SAXParserFactory** to retrieve an instance of the **SAXParser**.

```
SAXParser parser = factory.newSAXParser();
```

3. Create an instance of your class that extends **DefaultHandler**.

```
DefaultHandler handler = new SAXHandler();
```

4. Call the parser's **parse()** method, passing a reference to the XML data and the handler implementation class.

```
parser.parse(new File("garage.xml"), handler);
```


SAXGarage.java

```
import java.io.File;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import org.xml.sax.helpers.DefaultHandler;

public class SAXGarage {
   public static void main(String args[]) {
      if (args.length != 1) {
         System.out.println("Usage: java SAXGarage garage.xml");
         System.exit(1);
      }
      SAXParserFactory factory = SAXParserFactory.newInstance();

      try {
         SAXParser parser = factory.newSAXParser();
         DefaultHandler handler = new SAXHandler();
         parser.parse(new File(args[0]), handler);
      }
      catch (Exception e) {
         System.err.println ("ERROR " + e);
      }
   }
}
```

garage.xml

```
<?xml version="1.0"?>
<garage>
   <car miles="0">
      <make>Porsche</make><model>911</model><year>2001</year>
   </car>
   <car miles="250000">
      <make>VW</make><model>Beetle</model><year>1974</year>
   </car>
   <van miles="50000">
      <make>Ford</make><model>E350</model><year>2000</year>
   </van>
</garage>
```

Try It:

Compile *SaxGarage.java* and run it using *garage.xml* as the input file to list the make and model of each car or van.

SAX EVENT METHODS

- Developers usually focus on implementing the methods in the **ContentHandler** interface when extending **DefaultHandler**.

- **setDocumentLocator()** —This method is called to pass a **Locator** object to your application.

  - Use the **Locator** object to determine the current parsing location.

- **startDocument()**, **endDocument()** — These methods are called when the beginning or end of the XML document is encountered.

- **startElement()**, **endElement()** — These methods are called when encountering an open or close tag.

```
startElement(String uri, String localName, String qName,
   Attributes atts)
```

  - *uri* — the namespace name that is associated with this element.
  - *localName* — the tagname without any namespace prefix.
  - *qName* — the fully-qualified tagname of the element (prefix + *localName*).
  - *atts* — the list of attributes associated with this element.

- **characters()** — This method is called when character data is encountered.

```
characters(char[] ch, int start, int length)
```

  - *ch* — the character array that contains the actual character data that was found.
  - *start* — the starting point in the array.
  - *length* — the number of characters to read from the array.

SAXHandler.java

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

class SAXHandler extends DefaultHandler {
   private boolean printChars = false;

   public void startDocument() throws SAXException {
      System.out.println("Vehicles In My Garage");
   }
   public void endDocument() throws SAXException {
      System.out.println("\nGarage Door Closed");
   }
   public void startElement(String namespaceURI, String localName,
         String qName, Attributes atts) throws SAXException  {

      if (qName.equals("make")) {
         System.out.print("\n");
      }
      if (qName.equals("make") || qName.equals("model")) {
         System.out.print(qName + " : ");
         printChars = true;
      }
   }
   public void characters(char ch[], int start, int length)
         throws SAXException {

      if (printChars) {
         String s = new String(ch, start, length);
         System.out.println(s);
      }
      printChars = false;
   }
}
```

DefaultHandler provides stubs for all SAX event handler interfaces.

INTRODUCTION TO DOM

- DOM is a platform- and language-independent interface for accessing XML documents.

- The parser creates an internal, tree-like data structure containing objects that represent the various parts of the XML document.

  - The classes that make up the objects in the tree implement various interfaces specified by the World Wide Web Consortium (W3C).

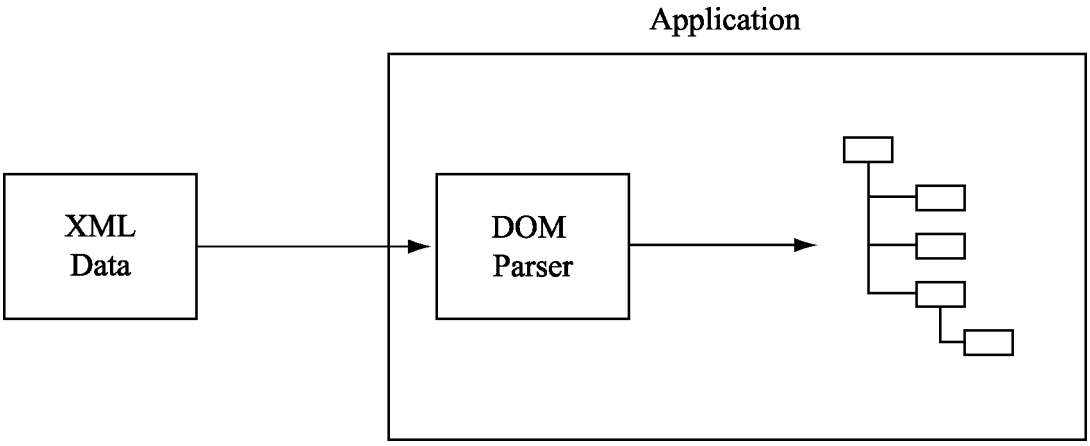
  - Your application can be DOM-compliant, instead of vendor-specific.

- After the parser is finished building the tree structure, it returns a reference to the top node of the tree, called the *Document*.

- An advantage of DOM is the ability to manipulate the document after it has been parsed.

  - You can also use DOM to create a new document from scratch.

- High memory use is a disadvantage of DOM when parsing large XML documents.


Application
XML
Data
DOM
Parser

PARSING DOM WITH JAXP

- Parsing an XML document using JAXP and DOM involves four steps:

1. Get an instance of the **DocumentBuilderFactory**.

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

2. Use the **DocumentBuilderFactory** to retrieve an instance of a **DocumentBuilder**.

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

3. Call the **parse()** method, passing a reference to the XML data.

```
Document doc = builder.parse(new File("garage.xml"));
```

4. Use the returned reference to the **Document** object to examine and manipulate the tree structure.

DOMGarage.java

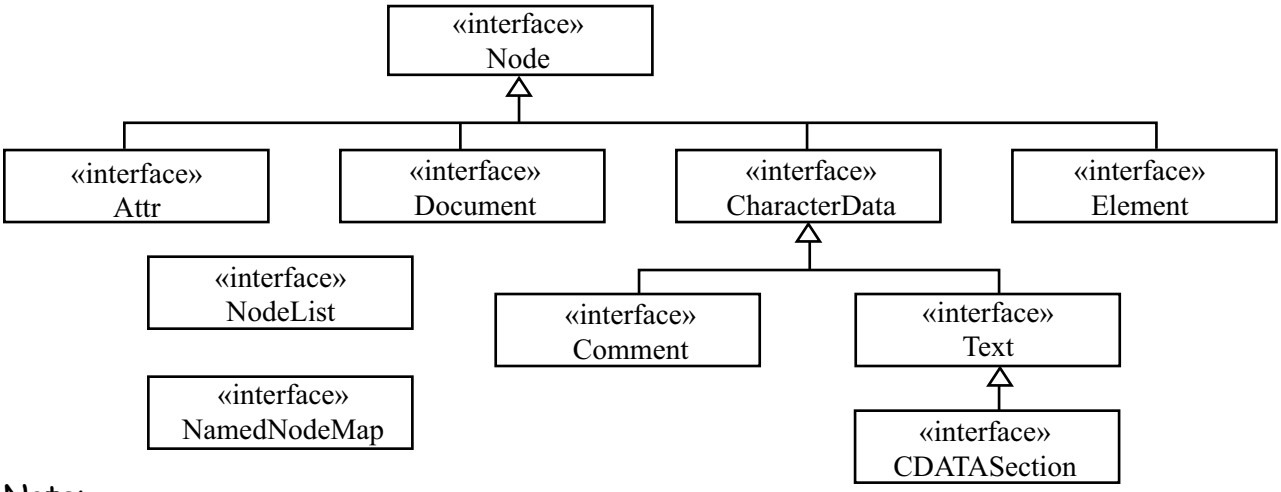
```
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class DOMGarage {
   public static void main(String args[]) {
      if (args.length!=1) {
         System.out.println ("Usage: java DOMGarage garage.xml");
         System.exit(1);
      }

      DocumentBuilderFactory factory =
         DocumentBuilderFactory.newInstance();

      try {
         DocumentBuilder builder = factory.newDocumentBuilder( );
         Document document = builder.parse( new File(args[0]) );
         System.out.println("Vehicles In My Garage\n");
         searchForVehicles(document);
         System.out.println("Garage Door Closed");
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
   ...
}
```

Try It:

Compile and run *DOMGarage.java* to list the make and model of each vehicle in the garage using a DOM parser.

THE DOM API

- DOM specifies twelve types of objects that can be included in the DOM data structure.
  - Each object represents something that exists in the XML document.
  - Each object implements an interface that is specific for that object.
  - All of the interfaces inherit from the **Node** interface.
- You can use **NodeList** and **NamedNodeMap** to store collections of **Node**s.
- The root node of the structure is the **Document** node.
- From the **Document** node you can traverse the structure by calling various methods.
  - **getDocumentElement()** returns a reference to the root element.
  - **getElementByTagName()** returns a **NodeList** of all elements with the given tag name.
- **Element** nodes represent the various elements in the structure.
  - You can append child nodes and manipulate attributes from the **Element** node.
- **Text** nodes are children of **Element** nodes and contain the actual text.



Note:
All interfaces are in the **org.w3c.dom** package.

DOMGarage.java

```
   ...
   public static void searchForVehicles(Document doc) {
      NodeList list = doc.getElementsByTagName("car");
      processVehicleList(list);
      list = doc.getElementsByTagName("van");
      processVehicleList(list);
   }
   public static void processVehicleList(NodeList autoList) {
      for (int i = 0; i < autoList.getLength(); i++) {
         Node auto = autoList.item(i);
         NodeList autoFeatures = auto.getChildNodes();

         for (int j = 0; j < autoFeatures.getLength(); j++) {
            Node featureNode = autoFeatures.item(j);

            if (featureNode.getNodeType() == Node.ELEMENT_NODE) {
               Element feature = (Element) featureNode;
               String name = feature.getNodeName();

               if (name.equals("make") || name.equals("model"))
                  System.out.println (name + ": " +
                     feature.getFirstChild().getNodeValue());
            }
         }
         System.out.println();
      }
   }
}
```

Element has various methods for querying node content.

VALIDATION

- There are two types of XML validation documents:

  1. Document Type Definitions (DTD) define the structure of the XML documents, but not the content.

  2. An XML Schema defines the valid structure, as well as content types for XML documents.

- The **ErrorHandler** interface provides three methods to deal with different types of errors.

  - **fatalError()** is called by the parser to report XML that is not well-formed.

  - **error()** is called to report that the XML document will not validate against the DTD or Schema.

  - **warning()** is called when the condition is not an error or fatal error.

- To validate with a DTD you must call **setValidating()** on the factory.

```
factory.setValidating(true)
```

- If you are validating against a schema you must instead call **setNamespaceAware()** and provide the schema document via the **setSchema()** method.

  - Use a **SchemaFactory** to retrieve a **Schema** object.

```
SchemaFactory f =
  SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = f.newSchema(new File("myschema.xsd"));
```


SAXValidator will validate an XML file against a DTD.

SAXValidator.java

```
...
public class SAXValidator {
   public static void main(String args[]) {
      if (args.length != 2) {
         System.out.println(
            "Usage: java SAXValidator xmlfile.xml schemafile.xsd");
         System.exit(1);
      }

      try {
         SchemaFactory schemaFactory =
          SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
         Schema schema = schemaFactory.newSchema(new File(args[1]));

         SAXParserFactory factory = SAXParserFactory.newInstance();
         factory.setNamespaceAware(true);
         factory.setSchema(schema);

         SAXParser parser = factory.newSAXParser();
         parser.parse(new File(args[0]), new SAXValidatorHandler());
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}

class SAXValidatorHandler extends DefaultHandler {
   private Locator locator;
   public void setDocumentLocator(Locator loc) {
      locator = loc;
   }
   public void error(SAXParseException e) {
      System.err.println("Validation error on line " +
         locator.getLineNumber() + ":\n" + e.getMessage());
   }
   public void fatalError(SAXParseException e) {
      System.err.println("Well-formedness error on line " +
         locator.getLineNumber() + ":\n" + e.getMessage());
   }
}
```

Pass object that implements the **ErrorHandler** interface to the **parse()** method.

Try It:

Run **SAXValidator** using *garage.xml* and then *badGarage.xml* passing in *garage.xsd* as the second parameter.

TRANSFORMATION

- *Transformation* is the process of editing an existing document with the goal of producing a different document.

- XSLT can transform an XML document into an HTML file, a text file, or another XML document.

  - XSLT is the W3C standard for creating XML documents that contain transformation templates.

  - The templates are applied to the data in an existing XML document to produce a new document.

- JAXP contains several interfaces and classes used to simplify the transformation.

  - Use **TransformerFactory** to create a **Transformer** object that contains the XSLT document.

```
TransformerFactory tFactory =
   TransformerFactory.newInstance();
Transformer transformer =
   tFactory.newTransformer(xslSource);
```

  - You use **Transformer**'s **transform()** method to create the new document.

```
transformer.transform(xmlSource, xmlResult);
```

    - **transform()** takes a **Source** and **Result** objects as paramters.

    - The **Source** and **Result** interfaces have implementations allowing you to work with DOM (**DOMSource**), SAX (**SAXSource**) or just regular streams (**StreamSource**).

GarageTransform.java

```
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

public class GarageTransform {
   public static void main(String[] args) throws Exception {
      Source xmlSource = new StreamSource("garage.xml");
      Source xslSource = new StreamSource("garage.xslt");
      Result xmlResult = new StreamResult("garage.html");

      TransformerFactory tFactory = TransformerFactory.newInstance();
      Transformer transformer = tFactory.newTransformer(xslSource);
      transformer.transform(xmlSource, xmlResult);
   }
}
```

garage.xslt

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      version="1.0">
   <xsl:template match="/">
      <html><body><div align="center">
         <table border="1">
            <xsl:apply-templates select="garage/*"/>
         </table>
      </div></body></html>
   </xsl:template>

   <xsl:template match="car|van">
      <tr>
         <td><xsl:value-of select="year"/></td>
         <td><xsl:value-of select="make"/></td>
         <td><xsl:value-of select="model"/></td>
      </tr>
   </xsl:template>
</xsl:stylesheet>
```

Try It:

Run *GarageTransform.java* and then open *garage.html* in a browser.

LABS

❶ Create an application that will use SAX to read through *garage.xml* and count the number of vehicles that have less than 20,000 miles.
(Solution: *MileageCounter.java*)

❷ Create an application that will parse *garage.xml* into a DOM structure. Create a new car **Element** node using **Document**'s **createElement()** method. Append the **Element** as a new child of the garage **Element**. Create three new **Element** nodes for **make**, **model,** and **year,** adding them as children of your new car **Element**. Create text nodes using **Document**'s **createTextNode()** method and add them as children to **make**, **model**, and **year** with appropriate data.
(Solution: *AddNewCar.java*)

❸ Modify your program from ❷ to send your DOM structure to a **Transformer** to run against *garage.xslt*, creating an HTML file. Open the HTML file in a browser to view your results.
(Solution: *AddNewCar2.java*)

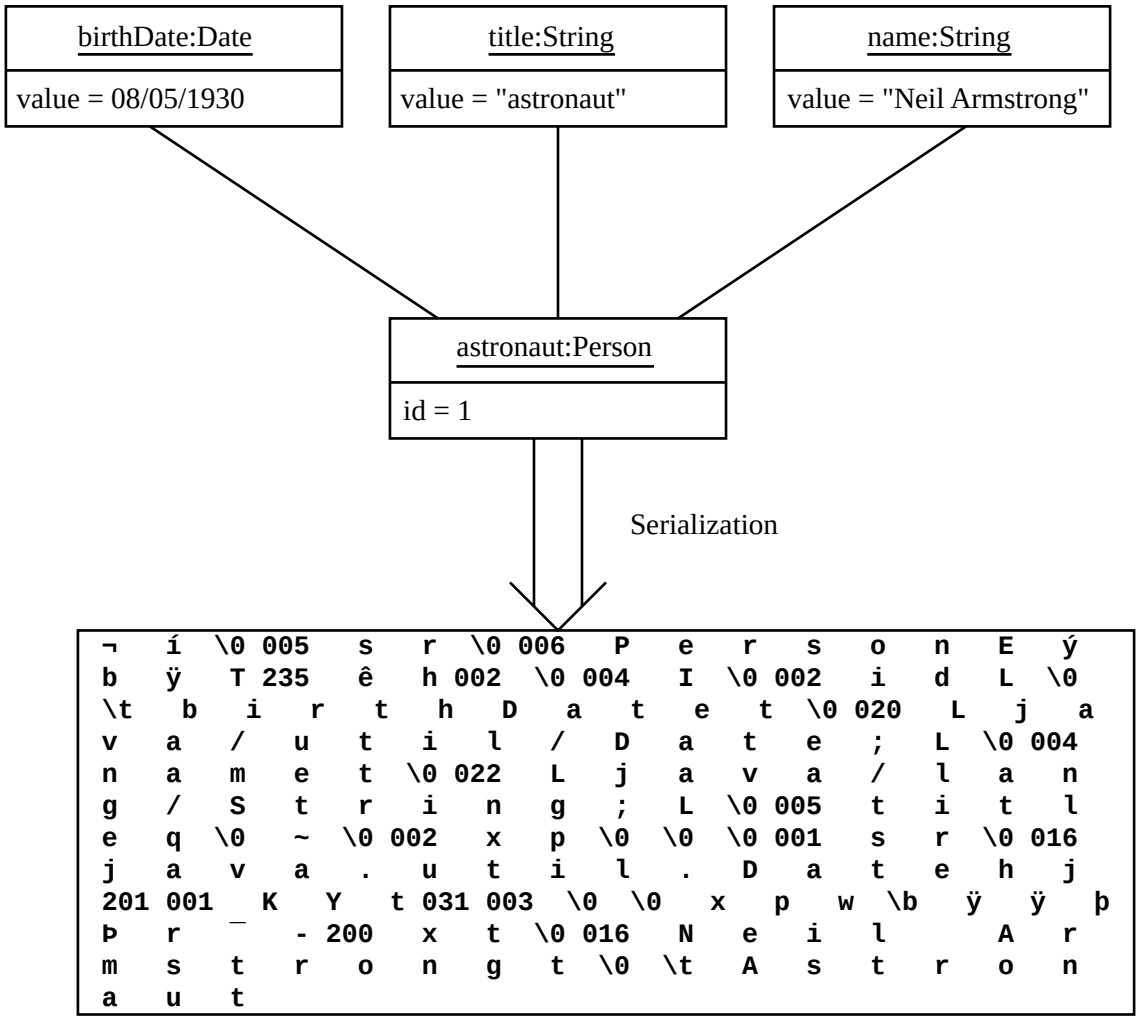
CHAPTER 5 - ADVANCED I/O – OBJECT SERIALIZATION

OBJECTIVES

- Save and retrieve objects.
- Send an object across a network connection.
- Customize the serialization process.
- Handle exceptions thrown during the serialization process.
- Support compatibility between versions of a **Serializable** class.

WHAT IS SERIALIZATION?

- *Serializing*, or *marshalling*, an object converts it to a "stream of data."
  - A serialized object contains the information necessary to deserialize, or unmarshal, the object.
- The **ObjectOutputStream** and **ObjectInputStream** classes are used to write and read objects on any type of existing I/O stream.
  - To write a serialized object to a file, create an **ObjectOutputStream** from a **FileOutputStream**.
    - The data in the file is in a proprietary Java format, so it will be difficult for you to read it.
  - To write a serialized object to another Java program, retrieve the **OutputStream** from a **Socket** and create the **ObjectOutputStream** from that.
    - The other Java program will re-create the object when reading it.
  - To write a serialized object to a **byte** array, create a **ByteArrayOutputStream** from a **byte[]** and wrap an **ObjectOutputStream** around it.
    - This array could then be stored in a DBMS.


birthDate:Date
value = 08/05/1930
title:String
value = "astronaut"
name:String
value = "Neil Armstrong"
astronaut:Person
id = 1
Serialization
¬ í \0 005 s r \0 006 P e r s o n E ý
b ÿ T 235 ê h 002 \0 004 I \0 002 i d L \0
\t b i r t h D a t e t \0 020 L j a
v a / u t i l / D a t e ; L \0 004
n a m e t \0 022 L j a v a / l a n
g / S t r i n g ; L \0 005 t i t l
e q \0 ~ \0 002 x p \0 \0 \0 001 s r \0 016
j a v a . u t i l . D a t e h j
201 001 K Y t 031 003 \0 \0 x p w \b ÿ ÿ þ
Þ r - 200 x t \0 016 N e i l A r
m s t r o n g t \0 \t A s t r o n
a u t

SERIALIZABLE OBJECTS

- A serializable object must be an instance of a class that implements **java.io.Serializable** (or **java.io.Externalizable**).

- **java.io.Serializable** defines an interface with no methods.

```
package java.io;
public interface Serializable {}
```

  - Implementing the interface simply marks your class for special treatment by the Virtual Machine.

- The serialization mechanism uses reflection to construct the **Serializable** objects.

  - If the superclass of a **Serializable** class is not **Serializable**, it must have a no-argument constructor.

    - The serialization mechanism uses this constructor to initialize the fields of the non-serializable superclass(es) of the object.

    - If the superclass does not have a no-argument constructor, you can use the **Externalizable** interface to work around this constraint.

This **Person** class is an example of a **Serializable** class. The **toString()** and **equals()** methods are included to compare original objects with restored objects.

Person.java

```
...
public class Person implements Serializable {
   private String name;
   private String title;
   private Date birthDate;
   private int id;

   public Person() {
   }
   public Person(String nm, String ti, String shortBirthDate, int i) {
      name = nm;
      title = ti;
      id = i;
      try {
         SimpleDateFormat df = new SimpleDateFormat("M/d/yyyy");
         birthDate = df.parse(shortBirthDate);
      }
      catch (ParseException e) {
         System.err.println("Parsing error: " + shortBirthDate);
         birthDate = new Date();
      }
   }
   public boolean equals(Object obj) {
      if (! (obj instanceof Person)) {
         return false;
      }

      Person p = (Person) obj;

      return name.equals(p.name) &&
         title.equals(p.title) &&
         birthDate.equals(p.birthDate) &&
         id == p.id;
   }
   public String toString() {
      SimpleDateFormat df = new SimpleDateFormat("MMM d, yyyy");
      return title + " " + name + " born " + df.format(birthDate) +
         " id " + id;
   }
   ...
}
```

WRITING AN OBJECT

- **ObjectOutputStream**'s **writeObject()** method serializes (marshals) an object.
  - The marshalled object contains structural information necessary to reconstruct the object.
- Build an **ObjectOutputStream** from any **OutputStream**.
- Marshalling an object involves dissecting the object into its component elements.
  - If the object has already been written to this **ObjectOutputStream**, only a handle is written.
    - This protects against recursive relationships and preserves shared references.
  - An **ObjectStreamClass** identifying the class of the object is written first.
    - This includes the name and **serialVersionUID** of the class.
  - Primitive fields are written to the stream.
  - Reference fields are serialized recursively.
  - **static** fields will not be serialized.
    - Only data specific to the instance is written.
  - **transient** fields will not be serialized.
- Since reference data (object members) must also be sent, they must implement **Serializable**.
- Transmitting an object will expose **private** data members.

The **writeObject()** method takes an **Object**, not a **Serializable**. This means that attempting to write a non-serializable object will not result in a compiler error, but in a **NotSerializableException**, which will be caught in the **IOException** catch, but will not be very descriptive.

WriteObj.java

```
import  java.io.FileOutputStream;
import  java.io.IOException;
import  java.io.ObjectOutputStream;

public class WriteObj {
   public static void main(String args[]) {
      Person armstrong = new Person("Neil Armstrong", "Astronaut",
         "08/05/1930", 1);

      ObjectOutputStream out = null;
      try {
         out = new ObjectOutputStream(
            new FileOutputStream("astronaut.ser"));
         out.writeObject(armstrong);
         System.out.println("Wrote \"" + armstrong + "\" to file");
      }
      catch (IOException e) {
         System.err.println("Error writing object: " + e.getMessage());
      }
      finally {
         try {
            out.close();
         }
         catch (IOException e) {
            System.err.println(e.getMessage());
         }
      }
   }
}
```

Try It:

Compile and run *WriteObj.java*. You can look at the resulting *astronaut.ser* file and see if you recognize any of the data.

READING AN OBJECT

- **ObjectInputStream**'s **readObject()** method deserializes (unmarshals) an object.
- Build an **ObjectInputStream** from any **InputStream**.
  - Creating an **ObjectInputStream** from a stream which does not contain a serialized object will result in a **StreamCorruptedException**.
- The **readObject()** method returns a **java.lang.Object**.
  - Downcast the object to the appropriate class, making sure to catch the **ClassCastException** which may result.
- Unmarshalling an object involves assembling the object from its component elements.
  - If the object has been read from the stream already, then a reference to the existing object is returned.
  - First, the **ObjectStreamClass** is read.
    - This is then used to load the class and verify the version.
  - The no-argument constructor of the first non-serializable superclass is called.
  - Primitive data members are set directly from the stream.
    - No **Serializable** class constructor is called and field initializers are also ignored.
  - Reference data members are deserialized recursively.

ReadObj.java

```
import  java.io.FileInputStream;
import  java.io.IOException;
import  java.io.ObjectInputStream;

public class ReadObj {
   public static void main(String args[]) {

      ObjectInputStream in = null;
      try {
         in = new ObjectInputStream(
            new  FileInputStream("astronaut.ser"));
         Person p = (Person) in.readObject();
         System.out.println(p);
      }
      catch (ClassCastException e) {
         System.err.println("Error casting object to a Person");
      }
      catch (ClassNotFoundException e) {
         System.err.println("Class not found");
      }
      catch (IOException e) {
         System.err.println("Error reading object: " + e.getMessage());
      }
      finally {
         try {
            in.close();
         }
         catch (IOException e) {
            System.err.println(e.getMessage());
         }
      }
   }
}
```

StreamCorruptedException extends **IOException**.

Try It:

Compile and run *ReadObj.java* to load the person.

HANDLING EXCEPTIONS

- Attempting to serialize an object containing data that is not **Serializable** will result in a **NotSerializableException**.

- Any **IOException** generated during an attempt to write an object to an **ObjectOutputStream** will be copied to the **OutputStream** in addition to being thrown.

- The application reading serialized objects is responsible for handling any exceptions discovered in the **InputStream**.

- The **readObject()** method throws a **ClassNotFoundException** if the VM can't find the object's *.class* file.

  - This happens even if you don't downcast the object.

BadBatchProcessor.java

```
...
public class BadBatchProcessor implements Runnable, Serializable {
   private ArrayList<Runnable> jobs;
   private ListIterator<Runnable> iterator;

   public BadBatchProcessor(ArrayList<Runnable> jobs) {
      this.jobs = jobs;
      iterator = jobs.listIterator();
   }
   ...
   public static void main(String args[]) {
      BadBatchProcessor processor = null;
      File serialFile = new File("batch_processor_bad.ser");
      if (serialFile.canRead()) {
         ObjectInputStream in = null;
         try {
            in = new ObjectInputStream(new FileInputStream(serialFile));
            processor = (BadBatchProcessor)in.readObject();
            System.err.println("Found previous batch processor : " +
                  processor);
         }
         catch (IOException ioex) {
            System.err.println("Error occurred reading "
                  + serialFile + ": " + ioex);
            System.exit(1);
         }
         catch (ClassNotFoundException cnfex) {
            System.err.println("Class " +   cnfex.getMessage()
                  + " not found while unmarshalling " + serialFile);
            System.exit(2);
         }
      ...
   }
}
```

The **ListIterator** implementation returned is not **Serializable**.

This will catch the exception thrown when reading the bad serialized object.

While serializing the **BadBatchProcessor**, an **Exception** will be thrown due to the non-serializable **ListIterator**.

Try It:

Run **BadBatchProcessor** once to create the serialized object with the **Exception**. Then run it again to see what happens when the object is read.

CUSTOMIZING SERIALIZATION

- Serializing an object also serializes the data members of the object.
- Use the **transient** modifier to specify that a data member should not be serialized.
  - The **transient** data will not be initialized when the object is read.
- Serialization behavior can be altered by defining the **readObject()** and **writeObject()** methods in the class to be serialized.

```
private void readObject(ObjectInputStream in)
  throws IOException, ClassNotFoundException

private void writeObject(ObjectOutputStream out)
  throws IOException
```

  - Implementing **writeObject()** and **readObject()** allows the class to pre- and post-process the serialization process, usually handling **static** or **transient** data members.
  - **ObjectOutputStream** provides **defaultWriteObject()** to serialize the object.
    - This method can only be called from **writeObject()** to write the **Serializable** portion of the object.
  - **ObjectInputStream** provides **defaultReadObject()** to deserialize the object.
    - This method can only be called from **readObject()** to read the **Serializable** portion of the object.

Use the **transient** modifier either for data which is temporary, such as mouse coordinates, or for data which cannot be successfully serialized. While the **Serializable** interface does not require any implementation, it is the developer's responsibility to ensure that the class and all of its data members are, in fact, **Serializable**. Failure will result in a runtime exception.

BatchProcessor.java

```
...
public class BatchProcessor implements Runnable, Serializable {
   private ArrayList<Runnable> jobs;
   private transient ListIterator<Runnable> iterator;

   public BatchProcessor(ArrayList<Runnable> jobs) {
      this.jobs = jobs;
      iterator = jobs.listIterator();
   }

   public void run() {
      while (iterator.hasNext() && ! Thread.interrupted()) {
         Runnable job = iterator.next();
         job.run();
      }
   }
   ...
   private void readObject(ObjectInputStream in)
         throws IOException, ClassNotFoundException {
      in.defaultReadObject();
      // get a list iterator to start at the next job
      int firstJob = in.readInt();
      iterator = jobs.listIterator(firstJob);
   }

   private void writeObject(ObjectOutputStream out)
         throws IOException {
      out.defaultWriteObject();
      // store the index of the next job for use at deserialization
      out.writeInt(iterator.nextIndex());
   }
   ...
}
```

Since the **ListIterator** is not **Serializable**, declare it **transient**.

Deserialize the non-**transient** fields first.

Read the index stored at serialization and get an **Iterator** to continue where we left off.

Serialize the non-**transient** fields first.

Store the index of the next job for use at deserialization.

Try It:

BatchProcessor has a **main()** method. The first time you run it, it creates a new batch of test jobs and runs for a couple of seconds. Each time you run it after that it will continue where it left off.

CONTROLLING SERIALIZATION

- Classes that need to completely control the serialization process can implement **java.io.Externalizable**, which extends **Serializable**.
- **Externalizable** declares two methods, **readExternal()** and **writeExternal()**.
- This interface completely replaces the default serialization behavior for assembling and disassembling the component elements.
  - An **Externalizable** class must provide a no-argument constructor which is invoked before **readExternal()**.
  - The **readExternal()** and **writeExternal()** methods must explicitly read and write any data necessary to reconstruct the object.
    - This includes fields from the class and all its superclasses.
  - **readObject()** and **writeObject()** will not be called on the class or any superclass, and **defaultReadObject()** and **defaultWriteObject()** are not available.
  - The **ObjectStreamClass** and internal handle are used in the same way as they are for **Serializable** classes.
- Use **Externalizable** when you need more control than **readObject()** and **writeObject()** provide.
  - Coordinating with the default serialization may be tedious; it might be easier to use **Externalizable**.
  - An **Externalizable** object can override the serialization behavior of its superclass.

Customer is a subclass of **Person**, so it inherits **Person**'s serialization behavior. For privacy, we want to leave the personal information out of the serialized objects. The easiest way to do this without changing **Person** is to use **Externalizable**.

Customer.java

```
...
public class Customer extends Person implements Externalizable {
   private int accountNumber;

   // required no-argument constructor for Externalizable mechanism
   public Customer() {
      // provide a dummy birth date
      // Person can't handle null
      setBirthDate(new Date());
   }

   public Customer(int accountNumber, String name, String title,
         String birthDate, int id) {
      super(name, title, birthDate, id);
      this.accountNumber = accountNumber;
   }

   // read Person.id and Customer.accountNumber
   public void readExternal(ObjectInput in) throws IOException {
      setId(in.readInt());
      accountNumber = in.readInt();
   }

   // write Person.id and Customer.accountNumber, we don't want to
   // expose the other fields of Person on the output stream.
   public void writeExternal(ObjectOutput out) throws IOException {
      out.writeInt(getId());
      out.writeInt(accountNumber);
   }
   ...
}
```

Here we must define appropriate defaults for the fields we will not be serializing. For most, **null** will suffice.

All other fields for **Person** have been defaulted by **Customer**'s no-argument constructor.

The only data we write from **Person** is the **id**.

Try It:

Test **Customer** with **WriteCust** and **ReadCust**.

VERSIONING

- Every **Serializable** class contains a **serialVersionUID**.
  - By default, this **long** value is calculated from the name and signature of the class and its fields and methods using the Secure Hash Algorithm.
  - You can provide your own **public static final long serialVersionUID** for backward compatibility.
    - Use the **serialver** utility (before you change the class) to find out the old version ID: **serialver Person**.
- You must ensure that the changes are both forwards and backwards compatible.
  - Adding, removing, or modifying methods is compatible, though you must consider your business logic.
  - Adding fields is compatible.
    - The new class must handle default values for fields missing from old objects.
    - The old class will ignore the unknown fields in new objects.
  - Removing fields is incompatible.
    - The old class might rely on non-default values from the fields missing in new objects.
  - You may need to use **readObject()** and **writeObject()** to ensure compatibility.

Person.java.new

```
...
public class Person implements Serializable {
   public static final long serialVersionUID = 5043296006500641384L;

   private String name;
   private String title;
   private Date birthDate;
   private int id;
   private String quote =
      "Be respectful to your superiors, if you have any. - Mark Twain";

   ...
   public String toString() {
      SimpleDateFormat df = new SimpleDateFormat("MMM d, yyyy");
      StringBuilder result = new StringBuilder();
      result.append(title).append("  ").append(name);
      result.append("  born  ").append(df.format(birthDate));
      result.append("  id  ").append(id);
      if (quote == null) {
         result.append(" old class version: no quote specified");
      }
      else {
         result.append(" quote ");
         result.append('"').append(quote).append('"');
      }
      return  result.toString();
   }
   ...
}
```

Here we handle objects serialized under the old version of the class.

Try It:

Make sure you have already run **WriteObj** with the original version of **Person**.

Make a backup copy of *Person.java*, then copy *Person.java.new* to *Person.java*. Compile *Person.java*, then run **ReadObj** to view the old serialized object with the new class.

Now run **WriteObj** to create a new serialized object. Restore *Person.java* from the backup, then compile it and run **ReadObj** to view the new object with the old class.

LABS

❶ Write an **Employee** class with appropriate fields, including a **hireDate**. Write a **Department** class which has an **Employee** for the manager. Write an application which creates a **Department** object and writes it to a file. (Hint: Use **SimpleDateFormat** to create a hire date.)
(Solution: *Employee.java, Department.java, WriteDept.java*)

❷ Write an application that reads a **Department** object from a file and displays it.
(Solution: *ReadDept.java*)

❸ Modify the solution to ❷ to read from a file that doesn't contain a serialized object.
(Solution: *ReadJunk.java, junk.txt*)

In the next set of exercises you will modify your **Employee** class to create a new version. You should save a copy of *Employee.java* and *dept.ser* before continuing.

❹ Run **serialver** on the **Employee** class and copy the value into a new **serialVersionUID** field in your **Employee** class. Then, add a new **Date** field, **departmentStartDate**, to **Employee**. Do not modify **toString()** to include this field yet. In your constructor(s), default the **departmentStartDate** to the **hireDate**. Make sure your new class works with the serialized objects you created in ❶; you should still be able to display the object using **ReadDept**.
(Solution: *Employee.java.4*)

❺ Use **SimpleDateFormat** to include the **departmentStartDate** in the **toString()** method. What happens when you run **ReadDept**?
(Solution: *Employee.java.5, explanation5.txt*)

❻ Fix your **Employee** class using **readObject** so that if the **departmentStartDate** was missing in the serialized object, it would default to **hireDate**. Verify that this works with **ReadDept**.
(Solution: *Employee.java.6*)

❼ Modify your application that writes the department so that it uses the new **Employee** class with a non-default **departmentStartDate**. After creating the new serialized object, restore the original version of **Employee** and verify that you can read the new serialized object with the old class.
(Solution: *WriteDept7.java*)

CHAPTER 8 - NETWORKING WITH SOCKETS

OBJECTIVES

- Write Java programs that act as clients.
- Use the **Socket** class to specify a host and port for a server.
- Send and retrieve data over a socket.
- Create an iterative server program.
- Communicate between a client and server.
- Access Web content through URLs.

CLIENTS AND SERVERS

- A *client* is any program that initiates communication with a server.
  - A Java client can be either an applet or an application.
- A client typically contains the user interface, but uses a server to interact with a database or some other resource.
  - The client may be unable to interact directly with the resource, or it may just be more efficient to have a single server program accessing the resource.
- A *server* program provides a service and waits for clients to "call."
  - Servers in Java are typically applications or servlets (Java applications attached to a web server).
- A server can even act as a client, requesting services from another server.
- Clients and servers communicate using many different mechanisms:
  - Remote Method Invocation (RMI) — networking is made to look like normal method calls.
  - Applet/Servlet communication — networking is handled by the web server.
  - Common Object Request Broker Architecture (CORBA) — networking is handled by an ORB (usually a separate process) and calls look like normal method calls.
  - Sockets — data is transmitted directly between two separate processes.

In this chapter you will create a client-side socket communications class that communicates through sockets with a server program (which you will also create). Provided is a GUI-based client program that will use your socket communications class. The name of the provided client program is **ClientGUI**. The class you create will be called **SocketHandle**r, with one **static** method named **echo()**.

PORTS, ADDRESSES, AND PROTOCOLS

- A client needs to know what host the server is on.
  - A *host* is a computer that can be identified by either an IP address or a hostname (which is converted to an address).
- A client also needs to know which process on the host is the actual server program.
  - Process IDs change every time a program starts, so each host has a set of numbers called *ports* which identify servers.
  - We say that a server "listens" on a specific port.
- A client needs to know how to communicate with the server.
  - A *protocol* defines the format of the communication (how much data is being sent, what comes first, etc.).
  - Two types of protocol in common use are *stream* and *datagram*.
    - The most common stream protocol, TCP, is the default because it does a lot of automatic error checking.
    - The most common datagram protocol, UDP, is used when speed is very important and errors do not matter.
- A client must know the host, port, and protocol of the server in order to find and communicate with it properly.

Try It:

Compile and run *ClientGUI.java*. When the send button is pressed, **ClientGUI** passes the data from the **sendArea** to **SocketHandler**, which currently just returns the data to **ClientGUI**, who displays it in the **receiveArea**. Look at the source code for both *ClientGUI.java* and *SocketHandler.java*. We will develop **SocketHandler** to send the data through sockets to a server, which will simply echo the data back, and then **SocketHandler** will return the echoed data to **ClientGUI**. We will not modify *ClientGUI.java*.

ClientGUI.java

```
...
public class ClientGUI extends JFrame {
   ...
      JButton send = new JButton("Send");
      send.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            String host = hostField.getText();
            int port = Integer.parseInt(portField.getText());
            String textToSend = sendArea.getText();

            String result = SocketHandler.echo(host, port, textToSend);
            receiveArea.setText(result);
         }
      });
   ...
}
```

SocketHandler.java

```
public class SocketHandler {
   public static String echo(String host, int port, String input) {
      return input;
   }
}
```

THE SOCKET CLASS

- Use a **java.net.Socket** object to connect to a server.
  - Various constructors are available to allow you to pass the server's hostname or IP address and the server's port number.

```
Socket s = new Socket(host, port);
```

    - Pass a host as either an **InetAddress** object or as a **String**.
    - Pass the port as an **int**.

- The **Socket** constructor locates the host on the network and connects to the server listening on the given port.
  - **UnknownHostException** means that the constructor cannot convert a hostname to an IP address.
  - **IOException** means there is a problem finding the host or port.

- The **Socket** constructors assume TCP protocol is being used.
  - Additional methods and classes exist for using UDP and other protocols.

Hands On:

Let's begin building *SocketHandler.java*.

SocketHandler.java

```
import java.io.*;
import  java.net.*;

public class SocketHandler {
   public static String echo(String host, int port, String input) {
      String receive = "";
      try {
         Socket s = new Socket(host, port);

      }
      catch (UnknownHostException e) {
         System.err.println("UnknownHost" + e.getMessage());
         System.exit(1);
      }
      catch (IOException e) {
         System.err.println(e.getMessage());
         System.exit(2);
      }
      return receive;
   }
}
```

COMMUNICATION USING I/O

- Once a client has constructed a **Socket** object, all communication between the client and server is done using the Java input and output classes.

- The **getOutputStream()** method returns the **OutputStream** object that you use to send data to the server.

- The **getInputStream()** method returns the **InputStream** object that you use to read data from the server.

  - You can convert these streams to any I/O class that you would like.

- Any type of data can be sent across these streams: **String** objects, Java primitives, or even **Serializable** objects.

  - You need to pick an I/O class that is appropriate for your data.

- Calling **close()** on the **Socket** will send an end-of-file to the server; do not forget this!

Hands On:

Add the I/O code to **SocketHandler**. The query string from the **ClientGUI sendArea** may have embedded newline characters, and each line will come back from the server individually, so we will send a "**\u0004**" and when it comes back we will know that everything sent has been echoed.

SocketHandler.java

```
        ...
        Socket s = new Socket(host, port);

        BufferedReader in = new BufferedReader(
           new InputStreamReader(s.getInputStream()));

        PrintWriter out =
           new PrintWriter(s.getOutputStream(), true);

        out.println(input);
        out.println("\u0004");

        String line = in.readLine();
        while (!line.equals("\u0004") ) {
           receive += line + "\n";
           System.err.println("Client DEBUG: " + line);
           line = in.readLine();
        }
        in.close();
        out.close();
        s.close();
     }
     catch (UnknownHostException e) {
     ...
```

Force the **PrintWriter** to flush the buffer.

We'll use this to end messages.

Stop when server replies "\u0004".

Try It:

Compile your client. If you are on a host or network with INET services (UNIX or Linux hosts will have these available) you can run your client against port 7, the standard echo server. Otherwise you will have to wait until you have built your server . . .

SERVERS

- Server programs are typically *daemons*: they stay running in the background all the time.
- Your server will loop infinitely, waiting for clients to try to connect it.
- When requests start coming in, you have two ways to handle the requests: iteratively or concurrently.
  - An *iterative* server simply handles each client completely before it goes on to listen for other clients.
  - A *concurrent* server handles multiple clients at once by using a separate thread or process to handle each one.
- You would use an iterative server when the service provided is quick, or the service is based on scarce resources.
  - An iterative server may be able to handle the client request quicker than it could handle the separate thread or process.
  - A scarce resource could be a database that only allows a few connections at once.
  - If more requests come in while the server is busy serving the previous one, they are queued by the underlying network software layer.

Hands On:

Let's begin building the server side. Create a class named **EchoServer** with a **main()** method. We will leave the **try** and **finally** blocks empty for now.

EchoServer.java

```
import java.io.*;
import java.net.*;

public class EchoServer {
   public static void main(String[] args) {
      Socket clientSocket = null;
      BufferedReader sockin = null;
      PrintWriter sockout = null;
      try {

      }
      catch (IOException e) {
         System.err.println(e.getMessage());
      }
      finally {

      }
   }
}
```

THE SERVERSOCKET CLASS

- Use a **ServerSocket** to listen for clients.
  - The port to be listened on is passed to the constructor.

```
ServerSocket ss = new ServerSocket(7777);
```

- The **accept()** method is a blocking call that returns a **Socket** object when a client connects with a **Socket**.

```
Socket s = ss.accept();
```

  - The **Socket** object returned by **accept()** has the client address, port, and protocol.
  - All client communication is done through this **Socket** object.
- Retrieve the input and output stream objects to send and receive data.
- Use the following steps for a basic iterative server:

1. Create a **ServerSocket** on a specific port.
2. Start an infinite loop.
   - 2a. Call **accept()**, which blocks and returns a **Socket** object.
   - 2b. Retrieve I/O stream objects from the **Socket** and communicate with the client.
   - 2c. Call **close()** to free up the temporary **Socket**.
3. Go back to the top of the loop.

Hands On:

Construct a **ServerSocket** on your own port. Ask the instructor for the appropriate port number. Create a **Socket** object when a client connects. Read each line from the client and echo it back through the socket.

EchoServer.java

```
...
public class EchoServer {
   public static void main(String[] args) {
      Socket clientSocket = null;
      BufferedReader sockin = null;
      PrintWriter sockout = null;
      try {
         ServerSocket listenSocket = new ServerSocket(7777);

         while(true) {
            clientSocket = listenSocket.accept();

            sockin = new BufferedReader(
               new InputStreamReader(clientSocket.getInputStream()));
            sockout =
               new PrintWriter(clientSocket.getOutputStream(), true);

            String linein;
            // Read from socket until client closes its end
            while ((linein = sockin.readLine()) != null) {
               sockout.println(linein);
               System.err.println("Server DEBUG: " + linein);
            }
            System.err.println("Server DEBUG: Connection closed");
            sockin.close();
            sockout.close();
            clientSocket.close();
         }
      }
      catch (IOException e) {
         System.err.println(e.getMessage());
      }
      finally {
         try {
            if (clientSocket != null)
               clientSocket.close();
         }
         catch (IOException e) {}
      }
   }
}
```

CONCURRENT SERVERS

- Concurrent servers are more complex to design, build, and maintain than iterative servers.

- Concurrent servers can handle multiple, simultaneous client requests.

  - They do this by executing multiple processes or threads, one for each client request.

- Concurrent servers can be faster than iterative servers if each client will take a long time to process, and if there are enough threads in the operating system for each client.

- The algorithm for a concurrent TCP server is:

  - Primary server (always running)

    1. Create a **ServerSocket** on the port for the service being offered.

    2. Block on an **accept()** message to the **ServerSocket**.

    3. Create an object (multi-threaded) or spawn a child process (multi-process) to handle the client request.

  - Secondary server (either an object or a separate process)

    1. Provide the service to the client via the **Socket** returned from the **accept()**.

    2. **close()** the **Socket** and clean up (a threaded object releases resources and is garbage collected, a process releases resources and dies).

ConcurrentServer.java

```
...
public class ConcurrentServer implements Runnable {
   private Socket client;
   private Thread theThread;
   private static int count;
   private static ThreadGroup threadGroup;

   public ConcurrentServer(Socket s) {
      client = s;
      theThread = new Thread(threadGroup, this, "Socket" + count);
      System.out.println("Client connected to server: " + count +
         ", Current active threads: " + threadGroup.activeCount());
      count++;
      theThread.start();
   }
   public void run(){
      BufferedReader sockin = null;
      PrintWriter sockout = null;
      try {
         sockin = new BufferedReader(
            new InputStreamReader(client.getInputStream()) );
         sockout = new PrintWriter( client.getOutputStream(), true);

         // loop, reading input from the socket and writing
         // the data to the client socket till client closes socket
         String linein = null;
         while( (linein = sockin.readLine()) != null ) {
            sockout.println( linein );
         }
      ...
   }
   public static void main(String[] args) {
      threadGroup = new ThreadGroup("Sockets");
      try {
         ServerSocket ss = new ServerSocket(7777);

         while(true) {
            // wait for the connection
            Socket s = ss.accept();
            // create a threaded object to handle the client
            new ConcurrentServer(s);
         }
      }
      ...
   }
}
```

Create a new thread.

Start the thread.

Get the input stream from the client socket.

THE URL CLASS

- A Uniform Resource Locator (URL) allows a Java program to easily retrieve data (typically a file) from a host, using well-known protocols.
  - The Java program is acting as a client.
  - The server you connect to depends on the protocol you specify.
    - Typical protocols for a URL include HTTP, FTP, file, and mailto.
- A URL can be constructed in much the same way you would type a URL into a web browser:

```
URL myPage = new URL("http://www.example.com/index.html");
```

  - The URL protocol, host, port, and file can also be specified individually:

```
URL myPage = new URL("http", "www.example.com", 80,
                     "index.html");
```

- Data can be retrieved from the **URL** object using one of three methods:
  - **getContent()** — requires a **ContentHandler** to convert the data.
  - **openStream()** — returns an **InputStream** that can be read.
  - **openConnection()** — returns a **URLConnection** object.

GetFile.java

```
import  java.io.BufferedReader;
import  java.io.InputStreamReader;
import  java.io.IOException;
import  java.net.MalformedURLException;
import  java.net.URL;

public class GetFile {
   public static void main(String[] args) {
      String contents = null;
      BufferedReader buf = null;
      try {
         URL myPage = new URL("http://www.google.com/index.html");

         buf = new BufferedReader(new InputStreamReader(
            myPage.openStream()));

         while ((contents = buf.readLine()) != null) {
            System.out.println(contents);
         }
      }
      catch (MalformedURLException e) {
         System.err.println(e.getMessage());
      }
      catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Note:

This program may fail to connect if you are inside a firewall without a configured proxy server. To specify proxy information, add the following two lines of code:

```
System.getProperties().put("proxyHost", "Host_IP");
System.getProperties().put("proxyPort", "Port_Number");
```

Replace ***Host_IP*** and ***Port_Number*** with the appropriate values for your proxy server.

THE URLCONNECTION CLASS

- The **URLConnection** class allows a program to obtain details of a resource prior to using it.

- A **URLConnection** object is retrieved from a URL object by calling **openConnection()**.

```
URLConnection conn = myPage.openConnection();
```

- Methods such as **getContentLength()** and **getContentType()** can be used to help determine what type of data the **URL** represents.

  - For instance, if the data is very long, you may want to create an input stream to read it, or if it is an image, you may not want to retrieve it into a **String**.

- **getInputStream()** and **getOutputStream()** can be used to read from and write to a **URLConnection**.

  - Writing to a **URLConnection** will only work if the protocol supports output (such as HTTP) and if **setDoOutput(true)** has been called.

    - This method is often used with applet-servlet communication.

GetConn.java

```
import  java.io.BufferedReader;
import  java.io.InputStreamReader;
import  java.io.IOException;
import  java.net.MalformedURLException;
import  java.net.URL;
import  java.net.URLConnection;

public class GetConn {
   public static void main(String[] args) {
      String contents = null;
      URLConnection conn = null;
      BufferedReader buf = null;
      try {
         URL myPage = new URL("http://www.google.com/index.html");

         conn = myPage.openConnection();

         System.out.println("File type: " + conn.getContentType());

         int len = conn.getContentLength();
         if (len == -1)
            System.out.println("File length unknown");
         else
            System.out.println("File length: " + len);

         System.out.println("Hit RETURN to continue");
         System.in.read();

         buf = new BufferedReader(new InputStreamReader(
            conn.getInputStream()));

         while ((contents = buf.readLine()) != null) {
            System.out.println(contents);
         }
      }
      catch (MalformedURLException e) {
         System.err.println(e.getMessage());
      }
      catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

LABS

❶ Create an application that takes the URL for an image on the command line and writes a local copy of the image. The program should verify that the URL actually points to an image using **URLConnection.getContentType()**. You can test your program with an arbitrary image from the internet using HTTP or FTP protocol, or with a file on the local machine using file protocol.
(Hint: The content-type header for most image types starts with **image/**.)
(Solution: *GetImage.java*)

❷ Create a server that waits for a connection and sends an arbitrary text file to any client that connects to it. (Hint: The server writes to the socket, but does not read from it.)
(Solutions: *SendFile.java*, *data.txt*)

❸ Write a client to access the server in ❷. (Hint: The client reads from the socket, but does not write to it.)
(Solution: *RetrieveFile.java*)

❹ Modify the server from ❷ so that it stores usage statistics, such as the total number of connections, in an object. Have the server listen on a separate administrative port and send the statistics object to the client. (Hint: The server must listen concurrently to both **ServerSocket** objects.)
(Solutions: *ServerStatistics.java*, *SendFile2.java*)

❺ Write an administrative client to read the statistics object from the server in ❹.
(Solution: *AdminClient.java*)

