

CONTENTS

Chapter 1 - Course Introduction	9
Course Objectives	10
Course Overview	12
Using the Workbook	13
Suggested References	14
Chapter 2 - Processing XML with Java — JAXP	17
The Java API for XML Processing	18
Introduction to SAX Parsing	20
SAXParser and JAXP	22
SAX Event Methods	24
Introduction to DOM	26
Parsing DOM with JAXP	28
The DOM API	30
Validation	32
Transformation	34
Labs	36
Chapter 3 - Introduction to Threads	39
Non-Threaded Applications	40
Threaded Applications	42
Creating Threads	44
Thread States	46
Runnable Threads	48
Coordinating Threads	50
Interrupting Threads	52
Runnable Interface	54
ThreadGroups	56
Labs	58
Chapter 4 - Thread Synchronization and Concurrency	61
Race Conditions	62
Synchronized Methods	64
Deadlocks	66

Synchronized Blocks	68
Synchronized Collections	70
Thread-Aware Collections	72
Thread Communication — wait()	74
Thread Communication — notify()	76
Executor	78
Callable	80
Labs	82
Chapter 5 - Advanced I/O — Object Serialization	85
What is Serialization?	86
Serializable Objects	88
Writing an Object	90
Reading an Object	92
Handling Exceptions	94
Customizing Serialization	96
Controlling Serialization	98
Versioning	100
Labs	102
Chapter 6 - Advanced I/O — NIO	105
The java.nio Package	106
Buffers and Channels	108
Buffer Implementations	110
Buffer Methods	112
ByteBuffer Methods	114
FileChannel	116
File Locking	118
MappedByteBuffer	120
Transferring Data Between Channels	122
Character Sets	124
Labs	126
Chapter 7 - Reflection	129
Introduction to Reflection	130
The Class Class	132
The reflect Package	134
Constructors	136
Fields	138

Methods	140
Exception Handling and Reflection	142
JavaBeans	144
Dynamic Programming	146
Labs	148
Chapter 8 - Networking with Sockets	151
Clients and Servers	152
Ports, Addresses, and Protocols	154
The Socket Class	156
Communication Using I/O	158
Servers	160
The ServerSocket Class	162
Concurrent Servers	164
The URL Class	166
The URLConnection Class	168
Labs	170
Chapter 9 - Remote Method Invocation	173
Distributed Applications	174
Stubs	176
Steps to Create a Remote Object	178
An RMI Client	180
An RMI Server	182
RMI Classes and Interfaces	184
Class Distribution	186
Parameter Passing and Serialization	188
Labs	190
Chapter 10 - Java Naming and Directory Interface (JNDI)	193
Naming and Directory Services	194
Namespaces and Contexts	196
Naming Operations	198
Bindings	200
Attributes	202
Directory Operations	204
DNS Lookups with JNDI	206
JNDI in Java EE	208
Labs	210

Chapter 11 - Java Performance Tuning	213
Is Java Slow?	214
Don't Optimize Until You Profile	216
HotSpot Virtual Machine	218
Garbage Collection Concepts	220
Garbage Collection Generations	222
Garbage Collection Algorithms	224
Object Creation	226
String, StringBuffer, and StringBuilder	228
Synchronized	230
Inline methods	232
Tuning Collections	234
Labs	236
Appendix A - Encryption with the javax.crypto Package	239
Cryptography Concepts	240
Encryption Keys	242
Cipher Algorithms	244
Modes and Padding Schemes	246
The Cipher Class	248
Encrypting and Decrypting Data	250
Cipher Output Stream	252
Cipher Input Stream	254
Encryption Using Password Ciphers	256
Exchanging Encrypted Keys	258
Sealed Objects	260
Labs	262
Appendix B - Native Methods	265
Overview of Java Native Methods and JNI	266
How to Create and Use Native Methods	268
Native Method Declaration	270
Using javah	272
Creating the Implementation Code	274
Compilation	276
Distribution	278
Using the Native Methods	280
JNI	282
Passing Arguments	284

Calling Java Methods in Native Code	286
JNI Signatures	288
Labs	290
Index	293

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

CHAPTER 1 - COURSE INTRODUCTION

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

COURSE OBJECTIVES

- * Access XML content with the Java API for XML Processing (JAXP).
- * Use threads to improve the responsiveness of your Java programs.
- * 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 sockets and Remote Method Invocation (RMI).
- * Bind and lookup objects in a naming service using the Java Naming and Directory Interface (JNDI).

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

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:** *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.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

- * The servlet container controls the life cycle of the servlet.
 - > When the first request is received, the container loads the servlet class
 - > The container uses a separate thread to call
 - > The container calls the destroy ()
- As with Java's finalize () method, don't count on this being called.
- * Override one of the init () methods for one-time initializations, instead of using a constructor.
 - > The simplest form takes no parameters.


```
public void init () {...}
```
 - > If you need to know container-specific configuration information, use the other version.


```
public void init (ServletConfig config) {...}
```
 - Whenever you use the ServletConfig approach, always call the superclass method, which performs additional initializations.


```
super.init (config);
```

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC

Topics are organized into first (*), second (>) and third (▪) level points.

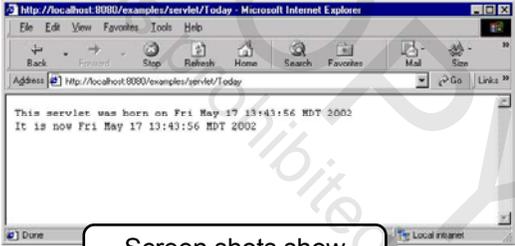
CHAPTER 2 **SERVLET BASICS**

Hands On:

Add an init () method to your Today servlet that initializes along with the current date:

```
Today.java
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
        ServletResponse response) throws ServletException, IOException
    {
        ...
        Servlet was born on " + bornOn.toString();
        " + today.toString();
    }
}
```

The init () method is called when the servlet is loaded into the container.



© 2002 ITCourseware, LLC Page 17

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

SUGGESTED REFERENCES

Bloch, Joshua. 2008. *Effective Java, 2nd Edition*. Pearson Education, Inc., Boston, MA. ISBN 860-1300201986.

Goetz, Brian. 2006. *Java Concurrency in Practice*. Addison-Wesley Professional, Upper Saddle River, NJ. ISBN 978-0321349606.

Harold, Elliott Rusty. 2013. *Java Network Programming, 4th Edition*. O'Reilly & Associates, Sebastopol, CA. ISBN 978-1449357672.

Horstmann, Cay and Gary Cornell. 2012. *Core Java 2, Volume I: Fundamentals, 9th Edition*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 978-0137081899.

Horstmann, Cay S. and Gary Cornell. 2013. *Core Java 2, Volume II: Advanced Features, 9th Edition*. Prentice Hall, Upper Saddle River, NJ. ISBN 978-0137081608.

Hunt, Charlie. 2011. *Java Performance*. Addison-Wesley Professional, Upper Saddle River, NJ. ISBN 978-0137142521.

Oaks, Scott. 2014. *Java Performance: The Definitive Guide*. O'Reilly & Associates, Sebastopol, CA. ISBN 978-1449358457.

<http://www.oracle.com/technetwork/java/index.html>

<http://www.javaworld.com>

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

CHAPTER 2 - PROCESSING XML WITH JAVA – JAXP

OBJECTIVES

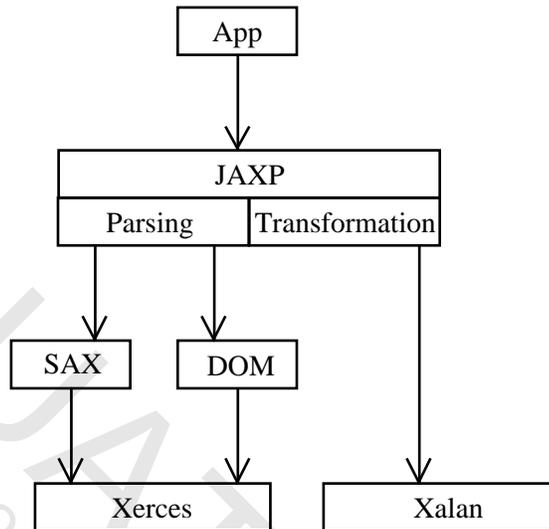
- * Describe the Java API for XML Processing (JAXP).
- * Use SAX callbacks to parse XML.
- * Traverse and manipulate the DOM tree.
- * Use JAXP to transform an XML document with XSLT.

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
```

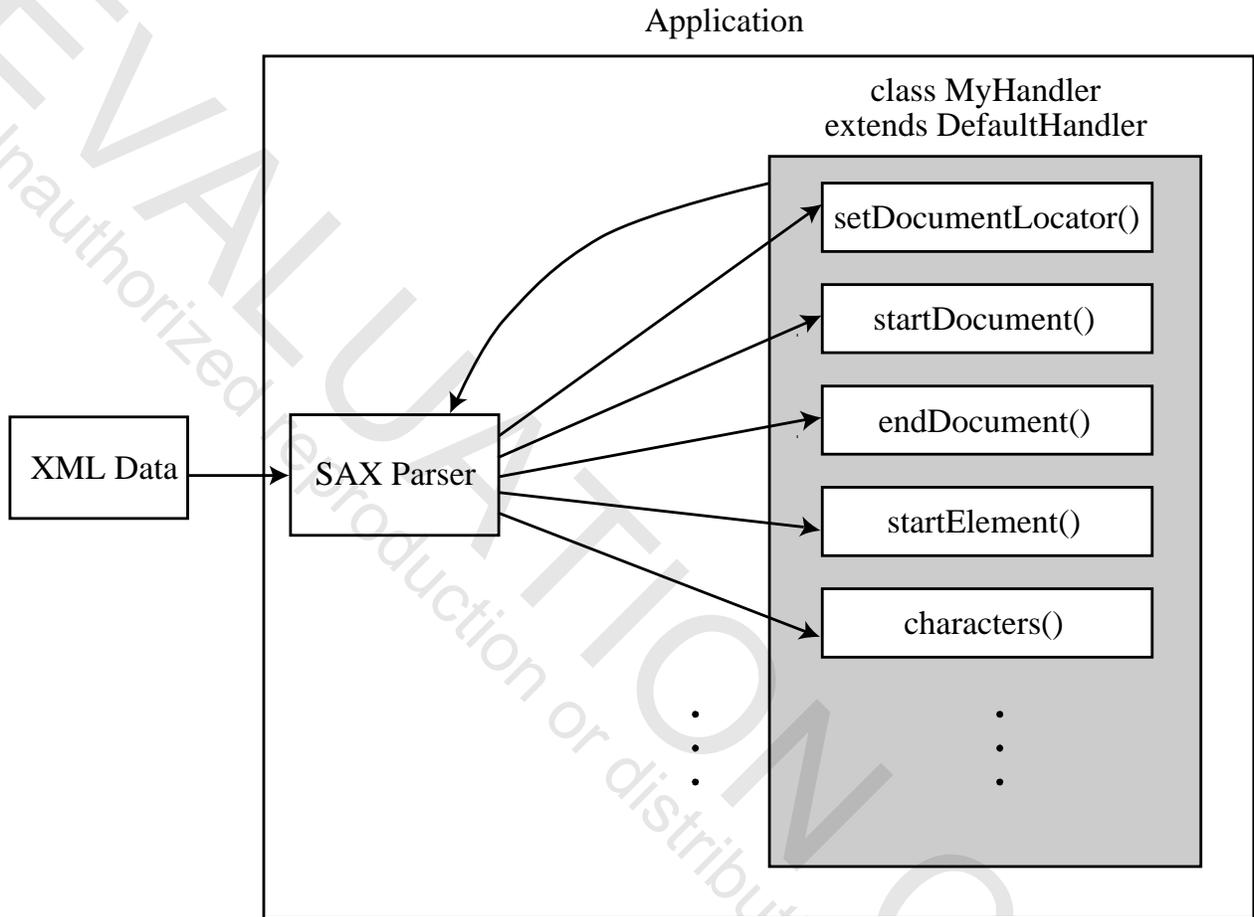
- * JAXP supports both object-based and event-based parsing of XML files.
 - Object-based parsing using the Document Object Model (DOM) involves the creation of an internal data structure to represent the XML document content.
 - Event-based parsing using Simple API for XML (SAX) treats the contents 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.



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

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.



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

```
package examples;

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

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

        SAXParserFactory factory = SAXParserFactory.newInstance();

        try {
            SAXParser parser = factory.newSAXParser();
            DefaultHandler handler = new SAXHandler();
            parser.parse(new File("garage.xml"), 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 and run *SaxGarage.java* 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

```
package examples;

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;
        }
    }

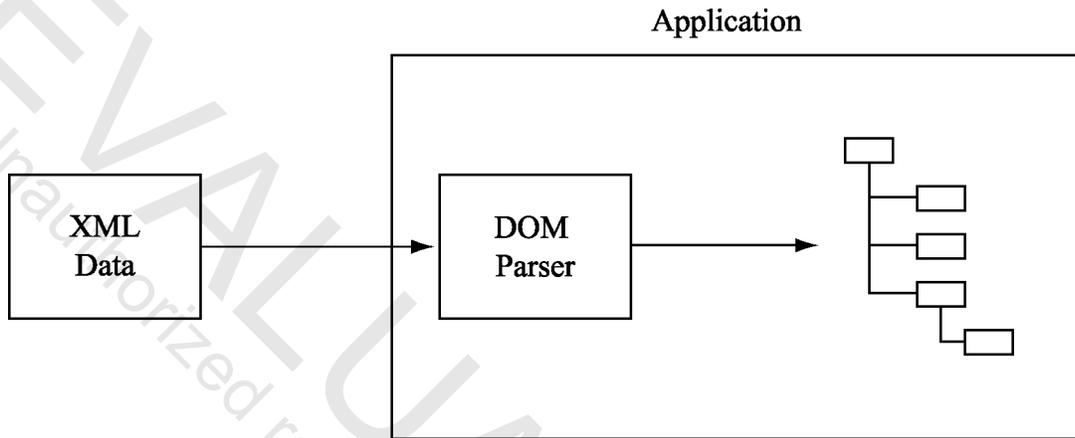
    public void characters(char ch[], int start, int length)
        throws SAXException {

        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.



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

```
package examples;

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

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

        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();

        try {
            DocumentBuilder builder = factory.newDocumentBuilder( );
            Document document = builder.parse( new File("garage.xml") );
            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 **Nodes**.
- * 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.

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

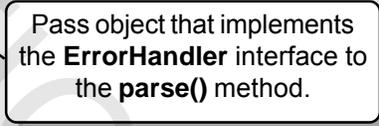
```
...
public class SAXValidator {

    public static void main(String args[]) {
        try {
            SchemaFactory schemaFactory =
                SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
            Schema schema = schemaFactory.newSchema(new File("garage.xsd"));

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

            SAXParser parser = factory.newSAXParser();
            parser.parse(new File("badgarage.xml"),
                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());
    }
}
}
```



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

```
package examples;

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.StreamResult;
import javax.xml.transform.stream.StreamSource;

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

        TransformerFactory tFactory = TransformerFactory.newInstance();
        Transformer transformer = tFactory.newTransformer(xsltSource);
        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 7 - REFLECTION

OBJECTIVES

- * Obtain the fields and methods of a class.
- * Dynamically invoke a method on an object.
- * Catch an exception thrown from a dynamically-invoked method.
- * Create a JavaBean.

INTRODUCTION TO REFLECTION

- * The Java Reflection API allows programs to find out everything about an object at runtime.
 - Since Java is a dynamic language, classes can be loaded, instantiated, and methods invoked on the objects after a program has started.
 - The Reflection API gives developers a window into what the JVM knows about an object.

- * The Reflection API is defined by the **java.lang.Class** class, and the classes and interfaces of the **java.lang.reflect** package.
 - With reflection, you can identify the type of an object, which class it extends, which interfaces it implements, and which modifiers, such as **public** or **final**, apply to the class.
 - You can query for the constructors, methods and fields of a class, and their modifiers.
 - You can instantiate a class without knowing its name at build time.
 - You can invoke methods or access fields of an object dynamically.
 - Unlike pointers in C++, you can not use reflection to bypass the visibility modifiers.

- * Reflection is primarily used by developers of tools, such as GUI design tools or debuggers.
 - You can use a **Method** object like a type-safe function pointer, but it may be better to design your calls to work with interfaces.

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

THE CLASS CLASS

* For any Java class, there is a corresponding **java.lang.Class** object.

* There are three ways to obtain the **Class** object:

1. Use the **class** literal.

```
Class<Person> personClass = Person.class;
```

2. Call the **forName()** static method of **java.lang.Class**.

```
Class<Person> personClass = Class.forName("Person");
```

3. Call the **getClass()** method of an object.

```
Person p = new Person("Sam", "Java Programmer",
    "03/20/92", 1);
Class<Person> personClass = p.getClass();
```

* The **Class** object provides information about the class.

- The name of the class.
- The class' constructors.
- The class' methods.
- The class' fields.

Person.java

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

    public Person(String nm, String ti, String bDate, int i) {
        name = nm;
        title = ti;
        id = i;
        try {
            SimpleDateFormat df = new SimpleDateFormat("M/d/yyyy");
            birthDate = df.parse(bDate);
        }
        catch (ParseException e) {
            System.err.println("Error parsing Birth Date: " + bDate);
            birthDate = new Date();
        }
    }
    public String getName() {
        return name;
    }
    public String getTitle() {
        return title;
    }
    public Date getBirthDate() {
        return birthDate;
    }
    public int getId() {
        return id;
    }
    public String toString() {
        SimpleDateFormat df = new SimpleDateFormat("MMM d, YYYY");
        return title + " " + name + " born " + df.format(birthDate);
    }
    public boolean equals(Object o) {
        if (o instanceof Person) {
            Person pTest = (Person) o;
            return name.equals(pTest.name) && title.equals(pTest.title)
                && birthDate.equals(pTest.birthDate) && id == pTest.id;
        }
        return false;
    }
}
```

THE REFLECT PACKAGE

* The **java.lang.reflect** package contains classes that encapsulate the parts of a class.

* The **Member** interface declares three methods of interest:

```
public Class getDeclaringClass()
```

```
public int getModifiers()
```

```
public String getName()
```

* The **Member** interface is implemented by the **Field**, **Constructor**, and **Method** classes.

- The **Field** class encapsulates information about a field, including a way to access the data.
- The **Constructor** class encapsulates information about an available constructor, including a way to invoke it.
- The **Method** class encapsulates information about a method, including a way to invoke it.

The **Member** interface also defines constants used when calling **SecurityManager**'s **checkMemberAccess()**.

Member.DECLARED identifies the set of members declared within a class or interface, not including inherited members.

Member.PUBLIC identifies the set of public members of a class or interface including inherited members.

CONSTRUCTORS

- * The `newInstance()` method of **Class** requires a default constructor.

```
Class<Person> personClass = Person.class;  
Person p = personClass.newInstance();
```

- * To instantiate an object with a different constructor, use a **Constructor** object.
 - The **Constructor** class provides a `newInstance()` method which takes an array of **Objects** for the parameter values.

CreateObject.java

```
...
public class CreateObject {
    public static void main(String[] args) {
        Class<Person> personClass = Person.class;

        Class<?>[] parmTypes = {String.class, String.class,
            String.class, int.class};

        Constructor<Person> pCreate = null;

        try {
            pCreate = personClass.getConstructor(parmTypes);
        }
        catch (NoSuchMethodException nsme) {
            System.err.println(nsme);
            System.exit(1);
        }

        Object[] parms = new Object[4];
        parms[0] = "Sam";
        parms[1] = "Java Programmer";
        parms[2] = "03/20/1992";
        parms[3] = 1;

        try {
            Person p = pCreate.newInstance(parms);
            System.out.println(p);
        }
        catch (InstantiationException ie) {
            System.err.println("Instantiation Exception " + ie);
        }
        catch (InvocationTargetException ie) {
            System.err.println("Invocation Target Exception " + ie);
        }
        catch (IllegalAccessException ie) {
            System.err.println("Illegal Access Exception " + ie);
        }
    }
}
```

The class literal works on primitives.

Try It:

Compile and run *CreateObject.java* to create a **Person** object, using **Constructor**'s **newInstance()** method.

FIELDS

* To find the **public** fields of a class, invoke the **getFields()** method on the **Class** object.

➤ The **getDeclaredFields()** method ignores inherited fields, but shows **public**, **protected**, and **private** members.

* Get a specific field with the **getField()** or **getDeclaredField()** methods.

```
Field nameField = personClass.getField("name");
```

➤ **getField()** throws a **NoSuchFieldException**, which must be caught or declared.

* Query or modify the value of the field with the **set()** and **get()** methods.

```
Person p1 = new Person("Sam");
String nm = (String) nameField.get(p1);
```

➤ **get()** and **set()** throw an **IllegalAccessException** if you don't have access to the field.

* The **getModifiers()** method returns an **int**, which defines the modifiers for a member.

➤ The **Modifier** class defines several **static final** fields and methods for decoding the modifier value.

The **ReadFields** application uses reflection to display the names and values of its public fields.

ReadFields.java

```
...
public class ReadFields {
    public String name;
    private String title;

    public ReadFields(String nm, String t) {
        name = nm;
        title = t;
    }

    public static void main(String[] args) {
        ReadFields[] test = new ReadFields[5];
        for (int i = 0; i < test.length; i++) {
            test[i] = new ReadFields("Object" + i, "Title" + i);
        }

        Class<ReadFields> rfClass = ReadFields.class;
        Field[] fields = rfClass.getFields();

        System.out.println("number of fields: " + fields.length);

        // display the name of the field and the value
        for (int i = 0; i < test.length; i++) {
            System.out.println("ReadFields #" + i);

            for (int j = 0; j < fields.length; j++) {
                try {
                    System.out.println("\t" + fields[j].getName() + ": "
                        + fields[j].get(test[i]);
                }
                catch (IllegalAccessException e) {
                    System.err.println(e);
                }
            }
        }
    }
}
```

The **getFields()** method will return all **public** fields of this class.

An **IllegalAccessException** will be thrown from the **get()** method if the underlying field is inaccessible.

Try It:

Run this program to display the **public** fields of an object. Modify the code to call **getDeclaredFields()** instead of **getFields()** to see all of the fields declared in the object.

METHODS

* To find the **public** methods of a class, invoke the **getMethods()** method on the **Class** object.

- The **getDeclaredMethods()** method ignores inherited methods, but shows **public**, **protected**, and **private** members.

* Get a specific method with **getMethod()** or **getDeclaredMethod()**.

```
Class[] parms = new Class[0];
Method method = personClass.getMethod("getName", parms);
```

- **getMethod()** throws a **NoSuchMethodException**, which must be caught or declared.
- If there are no parameters, you can substitute **null** for the empty array.

```
Method method = personClass.getMethod("getName", null);
```

* Invoke the method by providing an object and parameter array to the **invoke()** method:

```
Person p = new Person("Sam");
Object[] parms = new Object[0];
String nm = (String) method.invoke(p, parms);
```

- **invoke()** throws **IllegalAccessException** and **InvocationTargetException**, which must be caught or declared.

ReadMethods.java

```
...
public class ReadMethods {
    private String name;
    private String title;

    public String getName() {
        return name;
    }
    protected String getTitle() {
        return title;
    }
    private void setTitle(String ti) {
        title = ti;
    }
    public static void main(String[] args) {
        ReadMethods obj = new ReadMethods();
        Class<? extends ReadMethods> clazz = obj.getClass();
        Method[] methods = clazz.getMethods();

        System.out.println("Number of methods: " + methods.length);
        System.out.println(clazz.getName() + " methods:");

        for (int i = 0; i < methods.length; i++) {
            int mods = methods[i].getModifiers();
            if (Modifier.isStatic(mods))
                System.out.print("static ");

            System.out.print(methods[i].getReturnType().getName() + " ");

            Class<?> dcl = methods[i].getDeclaringClass();
            if (!clazz.getName().equals(dcl.getName())) {
                System.out.print(dcl.getName() + ".");
            }

            System.out.print(methods[i].getName() + "(");
            Class<?>[] parms = methods[i].getParameterTypes();
            for (int j = 0; j < parms.length; j++) {
                System.out.print((j > 0 ? ", " : "") + parms[j].getName());
            }
            System.out.println(")");
        }
    }
}
```

Loop through all methods.

Print **static** modifier.

Print return type.

Optionally print declaring class.

Print method name.

Print parameters.

EXCEPTION HANDLING AND REFLECTION

- * A dynamically-invoked method can throw an exception.
- * Use the **getExceptionTypes()** method to find which exceptions can be thrown from a method.

```
Class[] exceptions = method.getExceptionTypes();
```

- * If an exception is thrown by the invoked method, **invoke()** itself will throw an **InvocationTargetException**.
- * The **InvocationTargetException** will contain information about the originally-thrown exception.
 - Call **getCause()** to get the originally-thrown exception object.

This example creates a **Person2** object using a **Constructor**. The corresponding **Person2** constructor has been edited to throw a **ParseException**.

Person2.java

```
...
public class Person2 {
    ...
    public Person2(String nm, String ti, String bDate, int i)
        throws ParseException {
        ...
    }
}
```

CreatePerson.java

```
...
public class CreatePerson {
    public static void main(String[] args) {
        Class<Person2> clazz = Person2.class;
        Class<?>[] parmTypes = { String.class, String.class, String.class,
            int.class };

        try {
            Constructor<Person2> constructor = clazz
                .getConstructor(parmTypes);

            Object[] parms = { "Jane Doe", "CEO", "3/20/1964", 1 };
            Person2 person = (Person2) constructor.newInstance(parms);
            System.out.println(person);

            parms = new Object[] { "John Doe", "CIO", "xx/02/1968", 2 };
            person = (Person2) constructor.newInstance(parms);
            System.out.println(person);
        }
        catch (InvocationTargetException ite) {
            System.out.println("The invoked method threw an exception: "
                + ite.getCause().getClass());
            ite.printStackTrace();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Pass in an invalid
birthdate.

Try It:

Compile and run *CreatePerson.java* to test exception handling with reflection.

JAVABEANS

- * A JavaBean is a reusable Java component.
 - In component-based development, the idea is that you have solid, robust components that you can plug into an application to perform some functionality.
 - Application development then becomes more wiring together of components and less writing code from scratch.
 - An *integrated development environment* (IDE) allows developers to visually place and configure components.
- * The concept of component-driven development actually sounds a lot like Object-Oriented Development (OOD), and, in fact, JavaBeans are implemented as Java classes.
 - There are some special requirements of JavaBean classes to make it easier for IDEs to work with them.
 - A JavaBean class must implement **java.io.Serializable** and provide a no-argument constructor.
 - Properties are defined by **getPropertyName()** and **setPropertyName()** methods.
 - The JavaBean class should be packaged in a *.jar* and identified as a JavaBean.
- * If you follow the rules for JavaBeans, you can load your JavaBean into your IDE's toolbox, drag and drop it onto your window, set its properties, and even handle events.
 - The IDE performs all of those tasks by quering the bean class with reflection.

The **Person** class from earlier in the chapter almost fulfills all of the requirements of a JavaBean class. We need to add a no-argument constructor and have it implement **Serializable**. We should also provide set methods to match the get methods.

PersonBean.java

```
...
public class PersonBean implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private String title;
    private Date birthDate;
    private int id;

    public PersonBean() {
        name = "";
        title = "";
        birthDate = new Date();
    }
    ...
    public String getName() {
        return name;
    }
    public void setName(String nm) {
        name = nm;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String t) {
        title = t;
    }
    public int getId() {
        return id;
    }
    public void setId(int i) {
        id = i;
    }
    public String getBirthDate() {
        SimpleDateFormat df = new SimpleDateFormat("M/d/yyyy");
        return df.format(birthDate);
    }
    public void setBirthDate(String shortDate) throws ParseException {
        SimpleDateFormat df = new SimpleDateFormat("M/d/yyyy");
        birthDate = df.parse(shortDate);
    }
    ...
}
```

DYNAMIC PROGRAMMING

- * While the loading and execution of code in the JVM is dynamic, the actual programs themselves are static.
 - We have seen that it is possible to use reflection to dynamically create objects and invoke their methods, but the syntax is cumbersome; reflection really was not designed for that.
- * Tool developers can use reflection when developing IDEs.
 - IDE views that present your code as a tree structure made up of fields, constructors and methods are most likely using reflection to generate the tree.
 - Editors can use reflection to generate code completion drop-downs.
 - Debuggers can use reflection to show you the current state of fields in a running program.
- * Reflection is also used heavily in Java Enterprise Edition packages.
 - JavaServer Pages (JSP) use reflection to instantiate and access properties on server-side JavaBeans.
 - The JSP container calls the **newInstance()** method on the bean class to instantiate the bean, using its no-argument constructor.
 - The container converts property names into the appropriate **get()** or **set()** method call.
- * Enterprise Java Beans (EJB) use reflection in a similar way to create instances and identify properties.
- * JavaServer Faces (JSF) use reflection to dynamically populate beans with data that has been declared in an XML file.

EVALUATION COPY
Unauthorized reproduction or distribution is prohibited.

LABS

- ❶ The chapter directory contains a GUI program called *Reflect* that is a reflection test bed. All of the GUI part has been created (*ArgsDialog.java*, *ReflectFrame.java*), however the GUI calls methods in the **Reflect** class to find information about classes and objects. Currently the **Reflect** class contains stubs for the methods, but you need to fill in those stubs. In the **Reflect** class, fill in the code for the **getConstructors()** method to get a **List** of **Constructor** objects for the given class:

```
public List<Constructor<?>> getConstructors(String className)
```

You can test your program at any time by running *Reflect.java*.
(Solution: *Reflect1.java*)

- ❷ Write the code for the **instantiate()** method. Because we want to be able to create objects with overloaded constructors, use the **newInstance()** method in **Constructor**, not the one in **Class**.

```
public Object instantiate(Constructor<?> ctor, Object[] args)
```

(Solution: *Reflect2.java*)

- ❸ Write the code for the **getMethods()** method, which returns a **List** of **Method** objects:

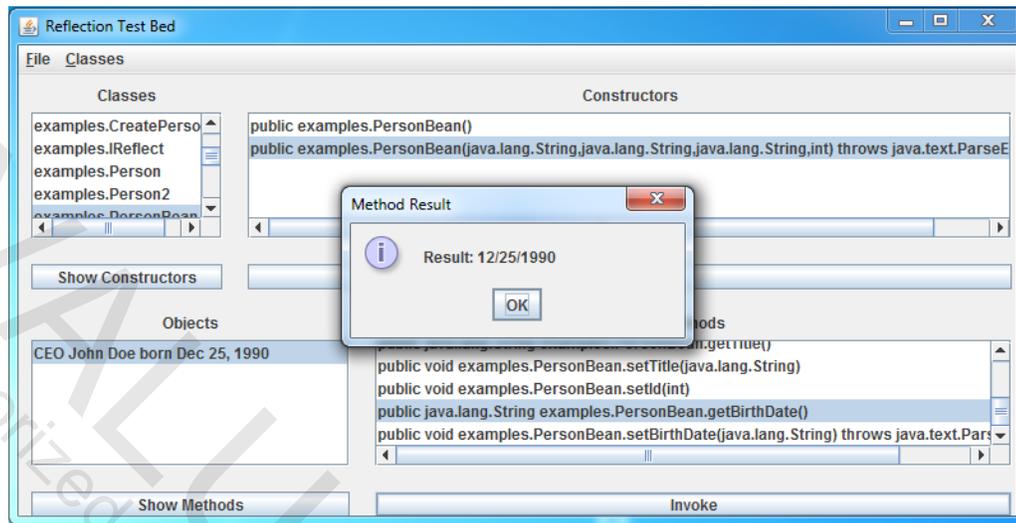
```
public List<Method> getMethods(Object obj)
```

(Solution: *Reflect3.java*)

- ❹ Write the code for the **invoke()** method to invoke the selected method on the selected object. The return value should be the return from the invoked method.

```
public Object invoke(Method method, Object target, Object[] args)
```

(Solution: *Reflect4.java*)

**Note:**

The GUI uses **JTable** components for data input. You must push 'enter' on your keyboard after you have entered a value into the **JTable** so the value is accepted.