**Purpose this doc**

1. **Bacis of jaxb**
2. **GenericJaxB Process was unable to treat special char only with spring utf-8 so you can define your own replacements of special charater that jaxB can use , it is also example of extending class in java-** EclipseLink/Release/2.4.0/JAXB RI Extensions/Character Escape Handler

**JAXB and Java IO (Files, Streams, Readers, and Writers)**

JAXB supports many different XML sources and targets.  In this post I'll give an overview of how JAXB can be used with those found in the*java.io* package: *File*,*InputStream*, *OutputStream*, *Reader*, *Writer* and their subclasses.

**Java Model**

The following domain model will be used for this post.

|  |
| --- |
| package blog.jaxb.io;    import javax.xml.bind.annotation.XmlRootElement;  import javax.xml.bind.annotation.XmlType;    @XmlRootElement  @XmlType(propOrder = {"street", "city"})  public class Address {        private String street;      private String city;        public String getStreet() {          return street;      }        public void setStreet(String street) {          this.street = street;      }        public String getCity() {          return city;      }        public void setCity(String city) {          this.city = city;      }    } |

**Marshalling an Object to XML**

The marshal operation is used to convert objects to XML.  In addition to producing the XML, a JAXB implementation will handle the escaping of special characters: " to &quot;, & to &amp; and < to &lt;. In this post we will focus on marshalling to *java.io.OutputStream*, and *java.io.Writer* (and their sub classes).  Below are examples of the marshal operation:  
  
**System.out**

|  |  |
| --- | --- |
| 1  2 | Marshaller marshaller = jaxbContext.createMarshaller();  marshaller.marshal(address, System.out); |

**OutputStream**  
A JAXB implementation will not close the OutputStream, this is something that you must do in your application code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Marshaller marshaller = jaxbContext.createMarshaller();  OutputStream outputStream = new FileOutputStream("output.xml");  try {      marshaller.marshal(address, outputStream);  } finally {      outputStream.close();  } |

**Writer**  
A JAXB implementation will not close the Writer, this is something that you must do in your application code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Marshaller marshaller = jaxbContext.createMarshaller();  Writer writer = new FileWriter("output.xml");  try {      marshaller.marshal(address, writer);  } finally {      writer.close();  } |

**Output**

|  |  |
| --- | --- |
| 1 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?><address><street>1 A Street</street><city>Any Town</city></address> |

**Writing into a Document (JAXB\_FRAGMENT)**

By default JAXB will include the XML header.  If you want to marshal contents into an XML stream that is also being populated from other sources you can tell JAXB to omit the header by setting the *Marshaller.JAXB\_FRAGMENT* property to true on the *Marshaller*.  This is a good example of why a JAXB implementation does not close streams.

|  |  |
| --- | --- |
| 1  2  3  4  5 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FRAGMENT, true);  System.out.print("<ROOT>");  marshaller.marshal(address, System.out);  System.out.print("</ROOT>"); |

**Output**

|  |  |
| --- | --- |
| 1 | <ROOT><address><street>1 A Street</street><city>Any Town</city></address></ROOT> |

**Formatting the Result (JAXB\_FORMATTED\_OUTPUT)**  
  
By default JAXB will not format the XML document.  This is done to save space, and not introduce any whitespace that may accidentally be interpreted as being significant.  To have JAXB format the output simply set the *Marshaller.JAXB\_FORMATTED\_OUTPUT* property to *true* on the *Marshaller*.

|  |  |
| --- | --- |
| 1  2  3 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  marshaller.marshal(address, System.out); |

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <address>      <street>1 A Street</street>      <city>Any Town</city>  </address> |

**Specifying the Encoding (JAXB\_ENCODING)**  
  
JAXB defaults the encoding to *UTF-8*.  You can specify an alternate encoding by setting the *Marshaller.JAXB\_Encoding* property on the *Marshaller*.

|  |  |
| --- | --- |
| 1  2  3  4 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  marshaller.setProperty(Marshaller.JAXB\_ENCODING, "ISO-8859-1");  marshaller.marshal(address, System.out); |

If you are marshalling to a Writer, you must ensure that you create the writer with the proper encoding type:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  marshaller.setProperty(Marshaller.JAXB\_ENCODING, "ISO-8859-1");  OutputStreamWriter writer = new OutputStreamWriter(System.out, "ISO-8859-1");  marshaller.marshal(address, writer); |

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5 | <?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>  <address>      <street>1 A Street</street>      <city>Any Town</city>  </address> |

**noNamespaceSchemaLocation (JAXB\_NO\_NAMESPACE\_SCHEMA\_LOCATION)**

We can specify the location of the XML schema corresponding to this document by setting the*Marshaller.JAXB\_NO\_NAMESPACE\_SCHEMA\_LOCATION* property on the *Marshaller*.

|  |  |
| --- | --- |
| 1  2  3  4 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  marshaller.setProperty(Marshaller.JAXB\_NO\_NAMESPACE\_SCHEMA\_LOCATION, "address.xsd");  marshaller.marshal(address, System.out); |

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <address      xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"      xsi:noNamespaceSchemaLocation="address.xsd">      <street>1 A Street</street>      <city>Any Town</city>  </address> |

**schemaLocation (JAXB\_SCHEMA\_LOCATION)**

If our document is namespace qualified then we can specify *the schemaLocation* by setting the*Marshaller.JAXB\_SCHEMA\_LOCATION* property on the *Marshaller*.

|  |  |
| --- | --- |
| 1  2  3  4 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  marshaller.setProperty(Marshaller.JAXB\_SCHEMA\_LOCATION, "<http://www.example.com/address> address.xsd");  marshaller.marshal(address, System.out); |

**Output**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0" encoding="UTF-8" standalone="yes"?>  <address      xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"      xsi:schemaLocation="<http://www.example.com/address> address.xsd">      <street>1 A Street</street>      <city>Any Town</city>  </address> |

**Extension Properties**

JAXB implementations may offer their own properties to offer capabilities beyond the specification.  EclipseLink JAXB (MOXy) for example offers a property that enables JSON binding.

|  |  |
| --- | --- |
| 1  2  3  4 | Marshaller marshaller = jc.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  marshaller.setProperty("eclipselink.media.type", "application/json");  marshaller.marshal(address, System.out); |

**Output**

|  |  |
| --- | --- |
| 1  2  3 | {"address" : {     "street" : "1 A Street",     "city" : "Any Town"}} |

**Unmarshalling an Object from XML**

The unmarshal operation is used to convert XML to objects.  In this post we will focus on unmarshalling from *java.io.File*,*java.io.InputStream*, and *java.io.Reader* (and their sub classes).  Below are examples of an unmarshal operation:

**File**

|  |  |
| --- | --- |
| 1  2  3 | Unarshaller unmarshaller = jaxbContext.createUnmarshaller();  File file = new File("input.xml");  Address address = (Address) unmarshaller.unmarshal(file); |

**InputStream**  
A JAXB implementation will not close the *InputStream*, this is something that you must do in your application code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Unmarshaller unmarshaller = jc.createUnmarshaller();  InputStream inputStream = new FileInputStream("input.xml");  Address address;  try {      address = (Address) unmarshaller.unmarshal(inputStream);  } finally  {      inputStream.close();  } |

**Reader**  
A JAXB implementation will not close the *Reader*, this is something that you must do in your application code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Unmarshaller unmarshaller = jc.createUnmarshaller();  Reader reader = new FileReader("input.xml");  Address address;  try {      address = (Address) unmarshaller.unmarshal(reader);  } finally  {      reader.close();  } |

**Handling XML Encodings**

**File**  
When unmarshalling from a *File*, there is nothing special you need to do to handle the encoding of the XML document.

|  |  |
| --- | --- |
| 1  2  3 | Unarshaller unmarshaller = jaxbContext.createUnmarshaller();  File file = new File("input.xml");  Address address = (Address) unmarshaller.unmarshal(file); |

**InputStream**

When unmarshalling from an InputStream, there is nothing special you need to do to handle the encoding of the XML document.:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Unmarshaller unmarshaller = jc.createUnmarshaller();  InputStream inputStream = new FileInputStream("input.xml");  Address address;  try {      Address address = (Address) unmarshaller.unmarshal(inputStream);  } finally  {      inputStream.close();  } |

**Reader**

When unmarshalling from a Reader you need to be aware of the XML encoding and configure the Reader accordingly:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Unmarshaller unmarshaller = jc.createUnmarshaller();  InputStream inputStream = new FileInputStream("input.xml");  Reader reader = new InputStreamReader(inputStream, "UTF-16");  try {      Address address = (Address) unmarshaller.unmarshal(reader);  } finally  {      reader.close();  } |

**Specifying the Type to Unmarshal**

If the root element of the XML document being unmarshalled does not match the one that you have associated with your class, then you can use one of the unmarshal methods that takes a Class parameter.  When unmarshalling from a File, InputStream, or Reader you need to wrap your input in a javax.xml.transform.stream.StreamSource:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | InputStream inputStream = new FileInputStream("src/blog/jaxb/io/input.xml");  Source source = new StreamSource(inputStream);  try {      Address address = (Address) unmarshaller.unmarshal(source, Address.class).getValue();  } finally  {      inputStream.close();  } |

**Extension Properties**

The JAXB specification does not define any Unmarshaller properties, however individual implementations may.  [EclipseLink JAXB (MOXy)](http://www.eclipse.org/eclipselink/moxy.php) for example offers a property that enables JSON binding.

|  |  |
| --- | --- |
| 1  2  3  4 | Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();  unmarshaller.setProperty("eclipselink.media.type", "application/json");  File file = new File("input.json");  Address address = (Address) unmarshaller.unmarshal(file); |

**Further Reading**

If you enjoyed this post, then you may also be interested in:

* [JAXB and Marshal/Unmarshal Schema Validation](http://bdoughan.blogspot.com/2010/12/jaxb-and-marshalunmarshal-schema.html)
* [Specifying EclipseLink MOXy as Your JAXB Provider](http://blog.bdoughan.com/2011/05/specifying-eclipselink-moxy-as-your.html)
* [JSON Binding with EclipseLink MOXy - Twitter Example](http://blog.bdoughan.com/2011/08/json-binding-with-eclipselink-moxy.html)

# Unicode Character 'RIGHT SINGLE QUOTATION MARK' (U+2019)

# EclipseLink/Release/2.4.0/JAXB RI Extensions/Character Escape Handler

< [EclipseLink](https://wiki.eclipse.org/EclipseLink)‎ | [Release](https://wiki.eclipse.org/EclipseLink/Release)‎ | [2.4.0/JAXB RI Extensions](https://wiki.eclipse.org/EclipseLink/Release/2.4.0/JAXB_RI_Extensions)

**Contents**

 [[hide](https://wiki.eclipse.org/EclipseLink/Release/2.4.0/JAXB_RI_Extensions/Character_Escape_Handler)]

* [**1** **Character Escape Handler**](https://wiki.eclipse.org/EclipseLink/Release/2.4.0/JAXB_RI_Extensions/Character_Escape_Handler#Character_Escape_Handler)
* [**2** **Behaviour**](https://wiki.eclipse.org/EclipseLink/Release/2.4.0/JAXB_RI_Extensions/Character_Escape_Handler#Behaviour)
* [**3** **Configuration**](https://wiki.eclipse.org/EclipseLink/Release/2.4.0/JAXB_RI_Extensions/Character_Escape_Handler#Configuration)
* [**4** **Appendix A - Example CharacterEscapeHandler**](https://wiki.eclipse.org/EclipseLink/Release/2.4.0/JAXB_RI_Extensions/Character_Escape_Handler#Appendix_A_-_Example_CharacterEscapeHandler)

# Character Escape Handler

In the current JAXB RI, developed by Sun, there is a series of "proprietary" JAXB extensions which provide advanced functionality outside of the JAXB specification (these extension classes and properties reside in the **com.sun.xml.bind** package).

The **CharacterEscapeHandler** interface provided in the Sun JAXB implementation allows users to write their own character escaping code, to be used when marshalling.

# Behaviour

If a custom **CharacterEscapeHandler** is set, then that class will be responsible for all character escaping during marshal operations.

Custom escape handling is not supported when marshalling to the following targets:

* javax.xml.stream.XMLStreamWriter
* javax.xml.stream.XMLEventWriter
* org.xml.sax.ContentHandler
* org.w3c.dom.Node

# Configuration

Your custom handler should be an implementation of the **org.eclipse.persistence.oxm.CharacterEscapeHandler** interface.

A custom **CharacterEscapeHandler** can be specified on a **Marshaller** using the setProperty() method:

...

JAXBContext ctx = ...

Marshaller m = ctx.createMarshaller();

m.setProperty(MarshallerProperties.CHARACTER\_ESCAPE\_HANDLER, new MyHandler());

...

**Note:** EclipseLink also supports Sun's CharacterEscapeHandler property names:

m.setProperty("com.sun.xml.bind.marshaller.CharacterEscapeHandler", new MyHandler());

m.setProperty("com.sun.xml.internal.bind.marshaller.CharacterEscapeHandler", new MyHandler());

# Appendix A - Example CharacterEscapeHandler

package example;

import java.io.IOException;

import java.io.Writer;

import org.eclipse.persistence.oxm.CharacterEscapeHandler;

public class MyHandler implements CharacterEscapeHandler {

/\*\*

\* Escape characters inside the buffer and send the output to the Writer.

\*

\* @exception IOException

\* In an error condition, IOException can be thrown to stop

\* the marshalling process.

\*/

public void escape(char[] buf, int start, int len, boolean isAttValue, Writer out) throws IOException {

for (int i = start; i < start + len; i++) {

char ch = buf[i];

if (ch == '&') {

out.write("&amp;");

continue;

}

if (ch == '"' && isAttValue) {

out.write("&quot;");

continue;

}

if (ch == '\'' && isAttValue) {

out.write("&apos;");

continue;

}

// you should handle other characters like < or >

// ...

if (ch > 0x7F) {

// escape everything above ASCII to &#xXXXX;

out.write("&#x");

out.write(Integer.toHexString(ch));

out.write(";");

continue;

}

// otherwise print normally

out.write(ch);

}

}

}