
# X-definition 4.1

## Java programming guide

**Václav Trojan**

trojan@syntea.cz

# Contents

# 1 Notice

*Questions, remarks, and bug reports please send to xdef@syntea.cz.*
*The actual version of X-definition you can download from http://www.xdef.cz*

# 2 Introduction

This text is an introduction to programming X-definitions. Part of this is a sample of practices on how to program typical tasks in Java. In several examples, derived from a simplified order processing task in a hypothetical system, we will show how to use the X definition definitions in Java. We assume that the reader is familiar with the basics of Java programming and is familiar with the XML language. We also assume that the reader has become familiar with X-definitions.

A detailed description of the X-definitions can be found will find in the following document at the Internet address:
http://xdef.syntea.cz/tutorial/en/userdoc/xdef4.1_eng.pdf.
The Java documentation is available at: http://xdef.syntea.cz/tutorial/en/userdoc/xdef.
A tutorial with which a reader can learn the use of X definitions can be found on
http://xdef.syntea.cz/tutorial/en.
The library for working with X definitions in Java is in the JAR file **syntea_xdef.jar**. It can be downloaded from: http://www.xdefinice.cz/en/download. The downloaded zip file also contains the source files of examples presented in this text.
An introduction to the construction process can be found on
http://xdef.syntea.cz/tutorial/en/userdoc/xdef4.4_construction_mode_eng.pdf
A description of the X components can be found on
http://xdef.syntea.cz/tutorial/en/userdoc/xdef4.1_X-component_eng.pdf

# 3 How to run X-definition

X-definition can be started in two different modes, either in a mode that validates and processes the input XML data according to the specified X definition (a particular model) or, using the specified model in the X-definition. The resulting XML document will be constructed according to this model.
In any case, we first have to compile the set of X-definitions and create the **XDPool** object. This object is reentrant (all values are constant after creation) and it is, therefore, appropriate to save this object in a static variable or "singleton" object. To run X definitions, create an **XDDocument** object from the XDPool object.

The result of running X definitions is either an XML element or a Java instance of the class in which the values of the processed XML document are accessible (the so-called **X-component**). H Reports of errors and statuses during the process are written into the so-called "**reporter**". The reporter can store the records either in working memory (org.xdef.ArrayReporter class) or in a file (class org.xdef.FileReporterWriter). In the methods to run X definitions, we can specify a reporter as a parameter. If the reporter is not specified (we will state "null"), then if bugs are reported during processing, the program will throw an exception in which errors are listed. If the reporter is specified, then you can test the presence of errors or warnings with methods "errors ()" or "ErrorWarnings ()"respectively.

## 3.1 Validation mode (and processing) of the input document

The validation and processing mode is run by the „xparse“ method on the instance of the XDDocument class. The input data is passed to the method by the first parameter. It is advisable to specify the reporter as the second parameter.

The program finds the appropriate model in the X definition (which must be declared "root" in the header of the X definition). Further processing takes place according to this model. During processing, the occurrence of

individual elements is checked by quantifiers and text values are checked using the appropriate validation method. When processing in different situations, the appropriate statements declared in the Script are called.

If an element is required, but it is not present in the input data, an error is reported and a command from the "onAbsence" section is called. If, on the other hand, the number of elements in the input data exceeds the maximum from the quantifier, an error is reported, and eventually, the "onExcess" section command is invoked. If the value of the data is found to be valid by the validation method, then, if the "onTrue" section is specified, the command of this section is called. If the data value is invalid, it will report an error and eventually, invokes the "onFalse" section command.

After reading all the attributes, the command "onStartElement" will be called (if specified). At the end of the element processing, the command is called "finally" (if specified).

Of particular importance is the "forget" section. This section does not contain any command, but it will cause the processed item to be released from the computer's memory. This allows you to process large data that would not come into processor working memory during processing.

## 3.2 Document construction mode

The document construction mode ("create mode") does not process an input XML document, but the resulting XML document is created according to the specified model. The construction mode is run by the "xcreate" method on the instance of the XDDocument class. By the first parameter of the xcreate method, the model according to which the resulting XML document will be constructed must be specified. Again, you should specify a parameter with a reporter.

The individual parts of the document being created are constructed according to the given model based on the so-called context. Context is data that is used at the time of creation to construct the target XML object. There is (or is not) any current context in the construction of each part of the document. If the "create" section command is specified in the Script of an XML model, then the value of a result of the "create" section command becomes the context for creating this XML object (this value must be usable as a context). This value then becomes the default for the construction of all descendants. The required type of this value varies according to the type of item being created. The commonly used method of the "create" section is an XPath expression that works with the XML document that has been assigned as the default context for the construction of a result (in this way you can "transform" an XML document from the context into the final form).
E.g. for text values and/or for attributes, the context value is converted to a string. This value is then set as the value of the created item. If the context value is "null" or an empty string, the attribute or text node is not constructed.

The context of elements can be a value of the following types:

- Element …… the result is made up of the content of this element. If the value is "null" then the element is not created.
- integer …… the number of elements corresponding to the minimum of this number and maximum in the quantifier is created.
- String …….. as a context, an element is created with a text node from the given value (ie, if the desired child is a text node, it is created from that string)
- Container …….. if the number of elements in the container is 0, nothing is created. Otherwise, elements are selected and this number of elements is created, corresponding to the minimum of the number of elements and the maximum in the section occurs.
- boolean …….. an element is created only if the value is true.
- null ..... the element is not created

Example with an explicit specification of the "create" section:

```
<A xd:script="create 1" x = "string; create 'abc'">
  optional string; create 'def';
</A>
```

The result will be:

```
<A x = "abc">def</A>
```

# 4 Tools for programming X-definitions in Java

## 4.1 Compilation of X-definition (creating XDPool object)

To work with X definitions, we will need an XDPool object (described by **org.xdef.XDPool**). XDPool is a binary object that is created by the compilation of X-definitions from source form. For example, the XDPool object can be created using the static **org.xdef.XDFactory.compileXD(...)** method. The first parameter of this method is the value of "java.util.Properties", where you can set some processing parameters (see ... XXXX). If this parameter is "null," the values set by the system (System.getProperties() method) are used. Other parameters then specify the X-definition sources to be compiled.

The XDPool object is reentrant (all data stored in it is constant). This means that one instance of this object can be used multiple times, or it can be shared in multiple processes. Therefore, you can save the created XDPool to a file and create it from the stored data again. This can be used for efficient programming: the XDPool object can be prepared separately from the source shape and saved to a file. You can then create the XDPool from this file (you do not need the source form of X-definitions and XDPool creation is then much faster). Another possibility is to generate the source of a Java class from the XDPool object using the "**XDFactory.genXDPoolClass**" method, so you can obtain the XDPool object any time from this class can then be part of a project distributed in source form.

*Note: In some cases, you need to create an XDPool object from a variety of sources (for example, a combination of source files stored in the database, from the Internet and the local file system, etc.). In this case, we must first create an object described by the interface **org.xdef.XDBuilder** uses the XDFactory class and uses it to incrementally create translation using setSource methods. Finally, we create the XDPool object using the **genXDPool** method called from XDBuilder. However, this procedure is in most cases not necessary, and it is usually provided with the static "compileXD" method from the XDFactory class. See Java documentation for XDBuilder class.*

## 4.2 Create the instance of XDDocument object

To work with X-definitions, use the XDPool object to create an instance of the XDDocument object that is used to work with the required XML document. You can create this object from the XDPool object by using the "**createXDDocument**" method. The parameter of this method specifies a specific default X-definition that is used for the validation or construction mode. The XDDocument is no longer reentrant, and its instance refers to the processed data. The result of the process of the XDDocument is an XML document (or an X component object). When processing the X definition, a protocol (error information, warnings, etc.) will also be generated. In the XDDocument object, the values of the variables declared in the X definitions are stored. In addition to the resulting XML document, it is also possible to obtain variable values from the XDDocument (using the „**getVariable**“ method). A protocol with the error reports, warnings, etc. is stored in a so-called **reporter** that is passed to the X-definition processor using the parameter of the startup method.

You can also set some values to the XDDocument before starting X-definition. The values of variables declared in X-definition can be passed using the **setVariable** method. The context that will be used in the construction mode can be set using the **setXDContext** method. By using the setUserObject method, you can set a user object that can be used in external user methods to link the program to external user data. The initial processing starts with xparse (XML document validation and processing) or xcreate (XML object construction).

To parse and validate XML data use the method **xparse**. To parse and validate JSON/XON data use method **jparse** and to parse and validate JSON/XON data use method **iparse**.

## 4.3 Other Java objects in X-Definitions

When the X-definition is processed it internally creates temporary control objects that relate to the currently processed part of the XML object. All of these objects are based on the common interface **org.xdef.proc.XXNode** (this also applies to XDDocument). The **org.xdef.proc.XXElement** object is created when processing the XML element and the **org.xdef.proc.XXData** object is created when processing the text values (ie, attributes or text nodes). These objects are created dynamically only during the processing, and they disappear after processing. However, they can be accessed in external "user" Java methods called from the Script. In these methods, it is often necessary to work with detailed information about the state of the processed objects or to influence them. Therefore, the object XXNode (or XXElement or XXData) that is created when processing the appropriate XML object can be passed as a parameter to an external method. If the external method requires it, this object can be passed to the external method (it must be always specified as the first parameter, but this parameter is not specified in the Script). It must be stated in the declaration of the relevant external method and the X-definitions automatically pass this value, if it is required in the external method).

The values of the parameters and variables declared in X definitions are represented by the interface **org.xdef.XDValue**. For common Java values (numbers, String, etc.), the external procedure parameters can be listed as a parameter list in the usual way, and the compiler automatically ensures that they are converted from the inner form to the Java object. The second way to pass parameters to external methods is to specify the parameter as an array of XDValue values. In this case, an array of objects with required values is passed to the method according to the list of parameters specified in a method call in the Script. Thus, one external method can be used from methods with a different number of parameters in the Script whose values are set as values of the array that is then passed to the method.

In some special cases, the program needs to know the values contained in the models in X definitions (such as occurrence limits, etc.). These objects are accessible via the interface **org.xdef.model.XMDefinition**, **org.xdef.model.XMElement** and **org.xdef.model.XMData**. They are
the objects corresponding to the models declared in the X-definitions, or parts thereof. Access to these objects is made possible from XXNode using the **getXMDefinition**, **getXMElement**, **getXMData** methods. The XDPool can be obtained using the **getXDPool** method.

Some processing parameters can be set using Properties. E.g. If "xdef_doctype" is set to "false," the X-definition does not allow the occurrence of "DOCTYPE," which is important from a security point of view, or the property "xdef_warnings" to "false" means the warning messages will not cause a program exception if reported. The language of report message by the property "xdef_language" More options see in **org.xdef.XDConstants**.
The error and information messages are passed to the X-definition processor by a reporter that allows you to write the processing protocol in the form of **org.xdef.sys.Report** objects to the files or to the memory of your computer.
The date and time values are implemented in the X definitions processor by the **org.xdef.sys.SDatetime** classes. For the intervals, the **org.xdef.sys.SDuration** class is used.
The class **org.xdef.sys.BNFGrammar** is used to work with the text values described by the extended BNF grammar.

# 5 Examples of how to work with XML data

In this chapter, we discuss some examples based on a specific hypothetical list of orders in the form of XML. Our task processes the input file of orders and checks their accuracy. In the examples, we describe both the source text of the Java program as well as the X-definitions and input data and are available in the functional form in the files that accompany this textbook, you can download this together with the X-definition Jar file at http://www.xdefinice.cz/en/download.

## 5.1 Task 1

Let's first check that the order structure is formally correct. The input data is in the "task1/ input /" directory. If no error is reported, the processed (checked) order will be stored in the "task1/output" directory. If an error is detected the file with errors will be stored in the "task1/errors /" directory.

The X-definitions and the Java source code used in each variant of this example are in the "src/task1 /" directory. In each of the variants of our example, we will show successive different ways of using X-definitions and supporting means.

### 5.1.1 Variant 1

Let's start with the simplest variant. Using X definition, we check the correctness of the input data, and if no error is found, we write the data into the output directory, otherwise we write the error log file (in text form) into the directory "task1/output/Order_1err.txt".
*Formally correct order is in the file "task1/Input /Order.xml":*

```
<Order Number="123" CustomerCode="ALFA">
  <DeliveryPlace>
    <Address Street="Oldroad" House="5" City="NewTown" ZIP="32321" />
  </DeliveryPlace>
  <Item ProductCode="0002" Quantity="2" />
  <Item ProductCode="0003" Quantity="1" />
</Order>
```

*X-definition describing the input data ("src/task1/Order1.xdef"):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Order" root="Order">

<Order Number="int" CustomerCode="string(1,20)">
  <DeliveryPlace>
    <Address Street="string(2,100)"
      House="int(1,9999)"
      City="string(2,100)"
      ZIP="num(5)"/>
  </DeliveryPlace>
  <Item xd:script="1..10" ProductCode="num(4)" Quantity="int(1,1000)"/>
</Order>

</xd:def>
```

When creating a program, we first need to obtain an XDPool object compiled from the source X-definitions. Because this object only contains constants, we can store it in a static final variable. From it, we create an XDDocument that we will need for further work. We also prepare a "reporter" to write the error log (the class ArrayReporter stores the log in the working memory of the computer). Using the "xparse" method, we start validation and processing the input data according to the X-definition. Then we will test whether any errors have been reported (i.e. whether the input data corresponds to the X-definition) and write the processed document into the appropriate directory (using the "KXmlUtils.writeXml" utility) if the input is correct or the error log file.

*Java source (the file „src /task1/Order1.java"):*

```
package task1;

import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;
import org.w3c.dom.Element;

public class Order1 {
        // Compile X-definitions to XDPool
        static final XDPool xpool = XDFactory.compileXD(null, "src/task1/Order1.xdef");

        public static void main(String[] args) throws Exception {
                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Order");

                // Create reporter
                ArrayReporter reporter = new ArrayReporter();

                // Run validation mode (you can also try task1/input/Order_err.xml)
```

```
                Element result = xdoc.xparse("task1/input/Order.xml", reporter);

                // Check if an error was reported
                if (reporter.errors()) {
                        // Print errors to the file
                        PrintStream ps = new PrintStream("task1/errors/Order_err.txt ");
                        reporter.printReports(ps);
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        // No errors, write the processed document to the file
                        KXmlUtils.writeXml("task1/output/Order.xml", result);
                        System.out.println("OK");
                }
        }
}
```

*The output of formally correct order will be in the file „task1/output/Order_1.xml“:*

```
<?xml version="1.0" encoding="UTF-8"?>
<Order CustomerCode="ALFA" Number="123">
  <DeliveryPlace>
    <Address City="NewTown" House="5" Street="Oldroad" ZIP="32321"/>
  </DeliveryPlace><Item ProductCode="0002" Quantity="2"/>
  <Item ProductCode="0003" Quantity="1"/>
</Order>
```

If the input data contains an error (in our case, we changed the "Quantity" attribute in the first element "Item" is not numeric, see file *task1/input/Order_err.xml*):

```
...
  <Item CommodityCode="0002" Quantity="xx"/>
…
```

then the error will be written in the file: *„task1/errors/Ordererr.txt“:*

```
E XDEF809: Incorrect value of 'int'; line=6; column=38;
source="file:/D:/cvs/DEV/java/examples/task1/input/Order_err.xml"; xpath=/Order/Item[1]/@Quantity;
X-position=Order#Order/Item/@Quantity
```

### 5.1.2 Variant 2: save compiled X definitions (XDPool) to a binary file

Let's show you how to proceed if we want to save the XDPool object created from the source X definition in a binary file and then use it. XDPool implements the interface Serializable, so it is possible to use ObjectOutputStream and ObjectInputStream to write and to read the compiled instance of XDPool. In the program below we will compile the X definition to the XDPool object and save it to the file "src/task1/Order1.xp".

*Java program „src/task1/Order1a_gen.java“ generates the binary file „src/task1/Order1a.xp“ from the X-definition:*

```
package task1;

import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.util.Properties;

public class Order1a_gen {

        public static void main(String[] args) throws Exception {
                // Compile XDPool from the X-definition
                Properties props = new Properties();
                XDPool xpool = XDFactory.compileXD(props, "src/task1/Order1.xdef");
                // Write XDPool to the file
                ObjectOutputStream outstr= new ObjectOutputStream(
                        new FileOutputStream("src/task1/Order1a.xp"));
                outstr.writeObject(xpool);
                outstr.close();
        }
}
```

The next program creates the XDPool object from this file. Of course, the time to create an instance of the XDPool object will be significantly shorter than compilation from the source X definition.

*Note that with each new version of X definitions, it is advisable to re-create a binary file from XDPool translated from the source code, since it may not be compatible with the actual version.*

*Java program „src/task1/Order1a.java“ reads XDPool from the file „src/task1/Order.xp“ which was generated by the program „Order1a_gen“:*

```
package task1;

import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.PrintStream;
import org.w3c.dom.Element;

public class Order1a {
        public static void main(String[] args) throws Exception {
                // Get XDPool object from the file
                ObjectInputStream in = new ObjectInputStream(
                        new FileInputStream("src/task1/Order1a.xp"));
                XDPool xpool = (XDPool) in.readObject();
                in.close();

                // Create instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Order");

                // Create reporter
                ArrayReporter reporter = new ArrayReporter();

                // Run validation mode (you can also try task1/input/Order_err.xml)
                Element result = xdoc.xparse("task1/input/Order.xml", reporter);

                // Check if an error was reported
                if (reporter.errorWarnings()) {
                        // Print errors to the file
                        PrintStream ps = new PrintStream("task1/errors/Order_err.txt");
                        reporter.printReports(ps);
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        // No errors, write the processed document to the file
                        KXmlUtils.writeXml("task1/output/Order.xml", result);
                        System.out.println("OK");
                }
        }
}
```

### 5.1.3 Variant 3: Creating an error file using an external method

Let's modify our example now by manipulating errors using external Java methods and not by methods in the X-script. These external methods will be invoked from the X-script, but the code will be in the Java class. Because the error document is generated by an external method, there is no need to specify the element model with errors in X-definition. The error data will be generated directly in the appropriate Java method. The price we pay for this is that the structure of the error document is not described in X-definition, and is dependent on the Java code.

Note that we added the declaration of the external methods used in the Script to X-definition. The methods that are called from the Script must be static and their parameters must match the types that are specified in the declaration section.

Since the external methods must be static, it is necessary to allow external methods to work with instances of objects that are passed by the "getUserObject" method. This object must be stored in the XDDocument before running X-definition of Java using the "setUserObject" method (in our example, we use the XML document to save the errors). External Java methods called from a script must have certain properties (they must be declared "static" and "public").

The first parameter of an external method declared in Java can be of the XXNode type. This parameter is passed automatically from the X-definition. Even if it is specified in Java code this parameter is not declared in

the method call in the Script (however, it is declared in the declaration of the Java external method in the declaration section). In our example, we need it to get the necessary data about the processing state, such as the current position of the input source being processed, but also to obtain the connected user object (using the „getUserObject“ method).

Types of other parameters in the call statement in the Script match the parameter types specified in the external method. The second parameter in our example corresponds to the error code number. The external Java method from this data creates an "Error" element and adds it as a child to the root element "Errors".

*JavapProgram (the file „src/task1/Order3.java“):*

```
package task1;

import org.xdef.sys.ArrayReporter;
import org.xdef.sys.SPosition;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import org.xdef.proc.XXNode;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class Order3 {

        public static void main(String[] args) throws Exception {
                // Create instance of XDDocument object (from XDPool)
                // (external method "err" called from the Script see below)
                XDPool xpool = XDFactory.compileXD(null, "src/task1/Order3.xdef");

                // Create instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Order");

                // Create reporter
                ArrayReporter reporter = new ArrayReporter();

                // Prepare XML element for recording of errors
                Element errors =
                        KXmlUtils.newDocument(null, "Errors", null).getDocumentElement();
                xdoc.setUserObject(errors);

                // Run validation mode (you can also try task1/input/Order_err.xml)
                xdoc.xparse("task1/input/Order.xml", reporter);

                // Check errors
                if (errors.getChildNodes().getLength() > 0) {
                        // Write error information to the file
                        KXmlUtils.writeXml("task1/errors/Order_err.xml", errors);
                        System.err.println("Incorrect input data");
                } else {
                        // No errors, write the processed document to the file
                        KXmlUtils.writeXml("task1/output/Order.xml", xdoc.getElement());
                        System.out.println("OK");
                }
        }
}
```

*X-definition (the file „src/task1/Order3.xdef“):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Order" root="Order">

<xd:declaration>
  external method void task1.Order3ext.err(XXNode, XDValue[]);
</xd:declaration>

<Order
  Number="int; onFalse err(1); onAbsence err(2);"
  CustomerCode="string; onAbsence err(3);">

  <DeliveryPlace xd:script="onAbsence err(11);">
    <Address Street="string(2,100); onFalse err(12); onAbsence err(13);"
      House="int(1,9999); onFalse err(14); onAbsence err(15);"
      City="string(2,100); onFalse err(16); onAbsence err(17);"
      ZIP="num(5); onFalse err(18); onAbsence err(19);"/>
  </DeliveryPlace>

  <Item xd:script="occurs 1..10; onAbsence err(21); onExcess err(22)"
    ProductCode="num(4); onFalse err(23); onAbsence err(24)"
    Quantity="int(1,1000); onFalse err(25); onAbsence err(26)"/>

</Order>
```

```
</xd:def>
```

*Java class with the external method "err" (the file „src/task1/Order3_ext.java“):*

```
package task1;

import org.xdef.sys.SPosition;
import org.xdef.XDValue;
import org.xdef.proc.XXNode;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class Order3ext {

        /** Add error item. */
        public static void err(XXNode xnode, XDValue[] params) {
                Document doc = ((Element) xnode.getUserObject()).getOwnerDocument();
                Element newElem = doc.createElement("Error");
                newElem.setAttribute("ErrorCode", params[0].toString());
                Element root = xnode.getElement().getOwnerDocument().getDocumentElement();
                newElem.setAttribute("Customer", root.getAttribute("CustomerCode"));
                SPosition pos = xnode.getSPosition();
                newElem.setAttribute("Line", String.valueOf(pos.getLineNumber()));
                newElem.setAttribute("Column", String.valueOf(pos.getColumnNumber()));
                doc.getDocumentElement().appendChild(newElem);
                xnode.clearTemporaryReporter(); // remove the error
        }
}
```

## 5.2 Task 2

This task is a modification of the previous one, but we now process not one order, but a file with many orders. Their number of order items is not limited and may not even come into the computer memory.
*The input file with orders „task2/input/Orders..xml“:*

```
<Orders id = "123456789">

<Order Number="123" CustomerCode="ALFA">
  <DeliveryPlace>
    <Address Street="LIBERECKA" House="5" City="LIBEREC" ZIP="32321"/>
  </DeliveryPlace>
  <Item ProductCode="0002" Quantity="2"/>
  <Item ProductCode="0003" Quantity="1"/>
</Order>

<Order Number="456" CustomerCode="BETA">
  <DeliveryPlace>
    <Address Street="NA SLUPI" House="9" City="PRAHA" ZIP="11000"/>
  </DeliveryPlace>
  <Item ProductCode="0019" Quantity="50"/>
</Order>

</Orders>
```

### 5.2.1 Varaiant 1: Using the "forget" command

We will use the "forget" command to solve this task. This command can be given in the Script of an element and means that it releases an element from memory after it has been processed. In our task, however, we must ensure that the element is written to the output file before releasing it. X-definition can handle this situation - we will connect the data stream to the XDDocument by using the method "setStreamWriter". This stream enables us to write the processed objects (before it is released from memory). In the first variant, we will leave writing the report again to the X-definition tool.

*X-definition (“src/task2/Orders1.xdef”):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Orders" root="Orders">

<Orders id="num(9)">
  <Order xd:script="occurs +; forget"
    Number="int"
    CustomerCode="string(1,20)">
    <DeliveryPlace>
      <Address Street="string(2,100)"
        House="int(1,9999)"
```

```
        City="string(2,100)"
        ZIP="num(5)"/>
    </DeliveryPlace>

    <Item xd:script="occurs 1..10" ProductCode="num(4)" Quantity="int(1,1000)"/>

  </Order>
</Orders>

</xd:def>
```

The continuous writing of the processed document into the output file is ensured by setting the "setStreamWriter" procedure. X-definition in this case continuously writes the processed elements to this output stream. With the "forget" command in the Script, we tell X-definition to release this element from memory at the end of the element processing. The error reports are written directly to the file (this is why this time we use the FileReportWriter class instead of the ArrayReporter class we used in the previous examples).

*Program „src/task2/Orders1.java“:*

```
package task2;

import org.xdef.sys.FileReportWriter;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class Orders1 {
        public static void main(String[] args) throws Exception {
                // Compile X-definitions to XDPool
                XDPool xpool = XDFactory.compileXD(null, "src/task2/Orders1.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Orders");

                // The file where to rite result
                File vystup = new File("task2/output/Orders.xml");
                OutputStream out = new FileOutputStream(vystup);
                xdoc.setStreamWriter(out, "UTF-8", true);

                // The file with errors
                File errors = new File("task2/errors/Orders_err.txt");

                // Prepare the error reporter (write error directly to the file)
                FileReportWriter reporter = new FileReportWriter(errors);

                // Run validation mode (you can also try task2/input/Order_err.xml)
                xdoc.xparse("task2/input/Orders.xml", reporter);

                // close the output stream.
                out.close();
                reporter.close();

                // Check if errors reported
                if (reporter.errorWarnings()) {
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK");
                }
        }
}
```

### 5.2.2 Variant 2: writing the correct orders and the wrong orders separately to the files

In this variant we will show the possibility to divide the output of the processed items into two files: in one we will write only correct orders and in the other, we will write the incorrect ones. We will also show you a more complicated use of external methods and a connected user object.

In X-definition, the "writeOrder" method is called in the Script of the "Order" element in the "finally" section. Instead of setting the automatic output of the processed input items into the output file, we will write the processed elements using this method. The errors will be written in the "err" method. Finally, we have to close the streams by the "closeAll" method, which closes the output files if a file is not empty.

In the declaration section, we declare three counters to which we set the initial value "0". Here we will store the number of errors and the number of correct orders. The "errCount" counter counts errors, and the error number is then stored in the "errCountOld" counter so that we can see if the new error has been found in the processed order (and therefore not written to the "output" stream). In the "count" counter we can see the number of the processed correct order items.

Both, the "output" and "error" streams are written using the "XmlOutStream" objects that have implemented resources for the continuous writing of XML objects. The element can be written as a whole by the writeElement method. The second option is to divide the writing of the element header by the "writeElementStart" method, and then to write the child nodes of the element by the "writeElement" or "writeText" methods, and finally to terminate the writing by the "writeElementEnd" method. The property of the "XmlOutStream" class is that the output file is created only in the first writing.

So, first, we need to write the root element header by the "writingElementStart" method, and at the end of the process to call the "writeElementEnd" method. The individual error items are written to the appropriate data streams by the "writeElement" method.

Note that we have included the text of the declaration section in the CDATA section so that we can freely use the "<", "&" characters, and we do not have to use alternate entities for these operators.

File names are passed to X-definition by the program using the external variables "outFile" and "errFile".

*X-definition ("src/task2/Orders2.xdef"):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Orders" root="Orders">

<xd:declaration>
<![CDATA[
  /* File names of files with constomers and commodity oodes. */
  external String outFile, errFile;
  int errCount = 0, errCountOld = 0, count = 0;
  XmlOutStream outStream, errStream;

  /* write file with errors */
  void err(int code) {
    if (errCount++ == 0) {
      errStream = new XmlOutStream(errFile);
          /* Document header and start of the root element */
      errStream.writeElementStart(new Element("Errors"));
    }
    Element e = new Element('Error'); /* Pripravime element. */
    /* Set attributes. */
    e.setAttr("Customer", xpath("/Orders/Order/@CustomerCode").toString());
    e.setAttr("ErrorCode", toString(code));
    e.setAttr("Line", toString(getSourceLine()));
    e.setAttr("Column", toString(getSourceColumn()));
    errStream.writeElement(e); /* Write the element with error information */
  }

  /* Write result */
  void writeResult() {
    if (errCount != errCountOld) { /* Error in order */
      errCountOld = errCount; /* Save number of errors, to know in new error occurred */
      return; /* write nothing */
    }
    if (count++ == 0) {
      /* this is the first item*/
      outStream = new XmlOutStream(outFile, "windows-1250"); /* Prepare data stream */
      outStream.writeElementStart(getRootElement()); /* Write document header and root element */
    }
    outStream.writeElement(getElement()); /* write the processed order */
  }

  /* Close the output stream and the file with errors */
  void closeAll() {
    if (errCount > 0) { /* check if errors were reported */
      errStream.writeElementEnd(); /* write root element end tag */
      errStream.close(); /* close the data stream */
    }
    if (count > 0) { /* Check if a correct order exists */
      outStream.writeElementEnd(); /* write root element end tag */
      outStream.close(); /* close the data stream */
    }
```

```
  }
]]>
</xd:declaration>

<Orders id="num(9); onFalse err(98)"
    xd:script="finally closeAll();">
  <Order xd:script = "occurs +; onAbsence err(99); finally writeResult()"
    Number="int; onFalse err(1); onAbsence err(2);"
    CustomerCode="string; onAbsence err(3);">
    <DeliveryPlace xd:script="onAbsence err(11);">
      <Address Street="string(2,100); onFalse err(12); onAbsence err(13);"
        House="int(1,9999); onFalse err(14); onAbsence err(15);"
        City="string(2,100); onFalse err(16); onAbsence err(17);"
        ZIP="num(5); onFalse err(18); onAbsence err(19);"/>
    </DeliveryPlace>
    <Item xd:script="occurs 1..10; onAbsence err(21); onExcess err(22)"
      ProductCode="num(4); onFalse err(23); onAbsence err(24)"
      Quantity="int(1,1000); onFalse err(25); onAbsence err(26)"/>
  </Order>
</Orders>

</xd:def>
```

*Program „src/task2/Orders2.java“:*

```
package task2;

import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;

public class Orders2 {
        public static void main(String[] args) throws Exception {
                // Compile X-definition to XDPool
                XDPool xpool = XDFactory.compileXD(null, "src/task2/Orders2.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Orders");

                // Set external variables
                xdoc.setVariable("outFile", "task2/output/Orders.xml");
                xdoc.setVariable("errFile", "task2/errors/Orders_err.xml");

                // Prepare reporter
                ArrayReporter reporter = new ArrayReporter();

                // Run validation mode (you can also try task2/input/Order_err.xml)
                xdoc.xparse("task2/input/Orders.xml", reporter);

                // Throw an exception if unexpected errors detected
                reporter.checkAndThrowErrors();

                // checck reported errors
                if (xdoc.getVariable("errCount").intValue() != 0) {
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK");
                }
        }
}
```

### 5.2.3 Variant 3: Using the external Java methods

In this variant, we will again show the possibility to divide the output into two files, wherein one we will write only the correct orders and in the other, we will write the wrong ones. We will also demonstrate a more complex use of external methods and a connected user object, in which we have programmed the methods from the previous sample using the external methods in Java.

X-definition records the output object and reports errors using methods that are programmed in the external class.

*X-definition (“src/task2/Orders3.xdef”):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Orders" root="Orders">

<xd:declaration>
  external method void task2.Orders3ext.writeOrder(XXNode);
```

```
  external method void task2.Orders3ext.err(XXNode, long);
</xd:declaration>

<Orders id="num(9); onFalse err(0)">
  <Order xd:script = "occurs +; onAbsence err(0); finally writeOrder(); forget"
      Number="int; onFalse err(1); onAbsence err(2);"
          CustomerCode="string; onAbsence err(3);">
    <DeliveryPlace xd:script="onAbsence err(11);">
      <Address Street="string(2,100); onFalse err(12); onAbsence err(13);"
        House="int(1,9999); onFalse err(14); onAbsence err(15);"
        City="string(2,100); onFalse err(16); onAbsence err(17);"
        ZIP="num(5); onFalse err(18); onAbsence err(19);"/>
    </DeliveryPlace>
    <Item xd:script="occurs 1..10; onAbsence err(21); onExcess err(22)"
      ProductCode="num(4); onFalse err(23); onAbsence err(24)"
      Quantity="int(1,1000); onFalse err(25); onAbsence err(26)"/>
  </Order>
</Orders>

</xd:def>
```

Instead of setting a continuous write of the processed document into the output file, we will write the processed elements using the "writeOrder" method in the external "Order2ext" class. If we store the external class instance into the "writer" variable and attach it using the "setUserObject" method to the XDDocument object. Finally, we close the write by the "closeAll" method, which terminates the write to both the output file and the error file. We do not prepare the reporter this time and we instead set null to the parameter of the "xparse" method. Therefore, when an error occurs which is not processed explicitly by the Script the program throws a RuntimeException.
If no exception is thrown we close the files (by the command "writer.closeAll ();") that we created and get the number of errors written by the user object (command "writeErrorsNumber()").

*Program „src/task2/Orders3.java“ :*

```
package task2;

import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;

public class Orders3 {
        public static void main(String[] args) throws Exception {
                // Compile X-definition to XDPool
                XDPool xpool = XDFactory.compileXD(null, "src/task2/Orders3.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Orders");

                // create instasnce of Orders3ext
                Orders3ext writer = new Orders3ext("task2/output/Orders.xml",
                        "task2/errors/Orders_err.xml");
                xdoc.setUserObject(writer);

                // Run validation mode (you can also try task2/input/Order_err.xml)
                xdoc.xparse("task2/input/Orders.xml", null);

                // close all streams
                writer.closeAll();

                if (writer.errNum() != 0) {
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK");
                }
        }
}
```

The external class for continuous writing ensures that the correct order is written to the recorder in the "_outputWriter" variable and the error information in the variable "_errorWriter". The writer constructor is created when the first order or error information is written. When creating a recorder, you must first write to the stream the XML header and start tag of the root element (element name and its attributes) using the "writeElementStart" method. The individual child elements are written using the "writeNode" method. In the end, we still have to write the root element's end tag by the "writeElementEnd" method (see the "closeAll" method in the Java source code.

*Java source of a class with external methods ( „src/task2/Orders3ext.java“):*

```
package task2;
```

```
import org.xdef.sys.SPosition;
import org.xdef.xml.KXmlUtils;
import org.xdef.xml.KXpathExpr;
import org.xdef.XDFactory;
import org.xdef.XDXmlOutStream;
import org.xdef.proc.XXNode;
import java.io.IOException;
import javax.xml.xpath.XPathConstants;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class Orders3ext {

        private final String _outputFile;       // Output file
        private final String _errorFile;        // Error file
        private XDXmlOutStream _outputWriter;   // writer for output
        private XDXmlOutStream _errorWriter;    // writer for errors
        private Document _errorDoc;             // XML document with errors
        private Document _objDoc;               // XML document with orders
        private final KXpathExpr _xpath;        // prepared xpath expression
        private int _errCount;                  // error counter
        private int _errCountOld;               // previous value of the error counter
        private int _count;                     // counter of correct orders

        // Create instance of this class.
        public Orders3ext(String outputFile, String errorFile) throws IOException {
                _outputFile = outputFile; // output file name.
                _errorFile = errorFile; // error file name.
                // writers will be created when an item to bew written occurs
                _errorWriter = _outputWriter = null;
                // Prepare XPath expression to get customer code from an order
                // Because of the command "forget" it will be in the processed document
                // only one (the actually processed) order.
                // This XPath expression will be executed when an error item is generated.
                _xpath = new KXpathExpr("/Orders/Order[1]/@KodZakaznika");
                // Clear counters
                _errCount = _errCountOld = _count = 0;
        }

        // Write the order (only if no error was reported
        public static void writeOrder(XXNode xnode) {
                // Get "User object" (i.e. the instance of this class).
                Orders3ext x = (Orders3ext) xnode.getUserObject();
                if (x._errCount != x._errCountOld) { // an error was reported?
                        // set old error counter (i.e. no errors reported for next item)
                        x._errCountOld = x._errCount;
                } else {
                        // No error reported, so writ the order the result.
                        Element el = xnode.getElement();
                        if (x._count == 0) { // check if nothig was written yeat
                                // Create writer a write the XML header and the root element
                                try {
                                        x._outputWriter =
                                                XDFactory.createXDXmlOutStream(x._outputFile,
                                                "windows-1250", true);
                                } catch (Exception ex) {
                                        throw new RuntimeException(ex.getMessage());
                                }
                                x._objDoc = el.getOwnerDocument();
                                x._outputWriter.writeElementStart(x._objDoc.getDocumentElement());
                        }
                        x._count++; // increase the counter of correct orders.
                        // write the processed order
                        x._outputWriter.writeNode(el);
                }
        }

        // Create the writeOrder and set the variable "_error".
        public static void err(XXNode xnode, long code) {
                // Get "User object" (i.e. the instance of this class).
                Orders3ext x = (Orders3ext) xnode.getUserObject();
                // Create the XML writer for errors (if it was not created yeat)
                if (x._errCount == 0) {
                        try {
                                x._errorWriter = XDFactory.createXDXmlOutStream(x._errorFile,
                                        "windows-1250",true);
                        } catch (Exception ex) {
                                throw new RuntimeException(ex.getMessage());
                        }
                        x._errorDoc = KXmlUtils.newDocument(null,"Errors",null);
                        // write XML header and the root element.
```

```
                        x._errorWriter.writeElementStart(x._errorDoc.getDocumentElement());
                }
                x._errCount++; // increase error counter
                // Create the element to be written.
                Element el = x._errorDoc.createElement("Error");
                el.setAttribute("ErrorCode", String.valueOf(code));
                String customer =
                        (String) x._xpath.evaluate(xnode.getElement(), XPathConstants.STRING);
                el.setAttribute("Customer", customer);
                SPosition pos = xnode.getSPosition();
                el.setAttribute("Line", String.valueOf(pos.getLineNumber()));
                el.setAttribute("Column", String.valueOf(pos.getColumnNumber()));
                x._errorWriter.writeNode(el);
        }

        // Get number of errors
        public int errNum() {return _errCount;}

        // Close created files
        public void closeAll() {
                // close result output stream (if something was written)
                if (_outputWriter != null) {
                        _outputWriter.closeStream(); // write root end tag and close the stream
                }
                // close error output stream (if something was written)
                if (_errorWriter != null) {
                        _errorWriter.closeStream(); // write root end tag and close the stream
                }
        }
}
```

## 5.3 Task 3

Now, let's check in the input data the correctness of the customer code and the commodity code according to the data stored in the external XML files. The input data is the same as in the first example.

*Input order „task3/input/Order.xml“:*

```
<Order Number="123" CustomerCode="ALFA">
  <DeliveryPlace>
    <Address Street="LIBERECKÁ" House="5" City="LIBEREC" ZIP="32321"/>
  </DeliveryPlace>
  <Item ProductCode="0002" Quantity="2"/>
  <Item ProductCode="0003" Quantity="1"/>
</Order>
```

*The file with customers „task3/input/Customers..xml“:*

```
<Customers>
  <Customer CustomerCode="ALFA">
     <Company Name="ALFA S.R.O." ID="1234578"/>
     <Address Street="KLADENSKÁ" House="5" City="KLADNO" ZIP="54321"/>
  </Customer>
  <Customer CustomerCode="BETA">
     <Company Name="BETA A.S" ID="1234580"/>
     <Address Street="MÌLNICKÁ" House="5" City="MÌLNÍK" ZIP="45321"/>
  </Customer>
</Customers>
```

*The file with products „task3/input/Products..xml“:*

```
<Products>
  <Product Code="0001" Name="bicycle" Price="150.50"/>
  <Product Code="0002" Name="motorcycle" Price="2340.00"/>
  <Product Code="0003" Name="car" Price="7777.00"/>
</Products>
```

### 5.3.1 Variant 1: automatic generation of error code

In the first variant, as usual, we demonstrate the simplest solution that generates the error code automatically. The XML data with customer codes and product numbers are passed through external variables. We get the customer code and product code in the script using the "xpath" method used in the validation methods "customerCode" and "productCode" (both declared in the declaration section). Note that the result of the method "xpath" is a Container. On Container objects, the "getLength" method returns the number of sequential elements in the object. We expect that if the value is correct, an appropriate element is found in the xpath argument. The value we look for using the xpath expression is obtained by the "getText" method, which

returns a string with the value of the currently processed attribute or the text node. We validate the value of the code with the declared methods. If the result is "true," the value is correct and if it is "false" the code value is incorrect. Note also that the method "error" used in validation methods puts the error message to the report file and returns the boolean value "false".

*X-defintioní „src/task3/Order1.xdef“:*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Order" root="Order">

<xd:declaration>
  /* Information about the customers and products. */
  external Element customers, products;
  /* Check if the customer exists in the customers list. */
  boolean customerCode() {
    if (xpath('Customer[@CustomerCode=&quot;' + getText() + '&quot;]',
                customers).getLength()!=0) return true;
    return error("Customer " + getText() + " not exists");
  }
  /* Check if the product exists in the products list. */
  boolean productCode() {
    if (xpath('Product[@Code=&quot;' + getText() + '&quot;]',
                products).getLength() != 0) return true;
    return error("Product " + getText() + " not exists");
  }
</xd:declaration>

<Order Number="int"
  CustomerCode="customerCode()" >

  <DeliveryPlace>
    <Address Street="string(2,100)"
      House="int(1,9999)"
      City="string(2,100)"
      ZIP="num(5)"/>
  </DeliveryPlace>

  <Item xd:script="occurs 1..10"
    ProductCode="productCode()"
    Quantity="int(1,1000)"/>
</Order>

</xd:def>
```

The Java program is virtually the same as in the first example, but we also have to set the external variables "products" and "customers" to the XDDocument. These variables are the Element type. The Java program is virtually the same as in the first example, but we also have to set the external variables "products" and "customers" to the XDDocument. These variables are also of the Element type. Note the "setVariable" method in Java code automatically converts a file into an Element.

*Program „src/task3/Order1.java“ :*

```
package task3;

import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;

public class Order1 {
        public static void main(String[] args) throws Exception {
                // Compile X-definitions to XDPool
                XDPool xpool = XDFactory.compileXD(null, "src/task3/Order1.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Order");

                // set variables "products" and "customers"
                xdoc.setVariable("products", "task3/input/Ptoducts.xml");
                xdoc.setVariable("customers", "task3/input/Customers.xml");

                // Prepare error reporter
                ArrayReporter reporter = new ArrayReporter();

                // Run validation mode (you can also try task3/input/Order_err.xml)
                xdoc.xparse("task3/input/Order.xml", reporter);

                // Check if an error was reported
                if (reporter.errorWarnings()) {
                        // Print errors to the file
                        PrintStream ps = new PrintStream("task3/errors/Order_err.txt");
```

```
                        reporter.printReports(ps);
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        // write porocessed document
                        KXmlUtils.writeXml("task3/output/Order_123.xml", xdoc.getElement());
                        System.out.println("OK");
                }
        }
}
```

The result of the program is similar to the first example.

### 5.3.2 Variant 2: declaration of the validation method

In this variant we make a change: we do the type checking by our validation method. The declared type is a method that - in our case - results in a boolean value. We will show that the error message can be done using the "error" method, which writes the error message into the reporter and returns the Boolean value "false". The Java program will be the same as in the previous variant, so let's just show the X-definition:

*X-definition „src/task3/Order2..xdef“:*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Order" root="Order">

<xd:declaration>
  /* Information about the customers and products. */
  external Element customers, items;

  boolean customer() {
    String s = getText();
    Container c = xpath('Customer[@CustomerCode="' + s + '"]', customers);
    return c.getLength()==0 ? error("Incorrect customer code: " + s) : true;
  }

  boolean item() {
    String s = getText();
    Container c = xpath('Product[@Code="' + s + '"]', products);
    return c.getLength()==0 ? error("Incorrect item code: " + s) : true;
  }
</xd:declaration>

<Order Number="int" CustomerCode= "customer()">

  <DeliveryPlace>
    <Address Street="string(2,100)"
      House="int(1,9999)"
      City="string(2,100)"
      ZIP="num(5)"/>
  </DeliveryPlace>

  <Item xd:script="occurs 1..10"
    ProductCode="item()"
    Quantity="int(1,1000)"/>
</Order>

</xd:def>
```

In the event of an error, the error message log will contain information with the text in the "error" method:

*The file with the error information is saved in „task3/errors/Order_123err.txt“ :*

```
Incorrect customer code: AGFA; line=2; column=35;
source='file:/D:/cvs/DEV/java/examples/task3/input/Order_err.xml'; pos=82;
xpath=/Order/@CustomerCode; X-position=Order#Order
E XDEF515: Value error; line=2; column=35;
source="file:/D:/cvs/DEV/java/examples/task3/input/Order_err.xml"; xpath=/Order/@CustomerCode; X-
position=Order#Order/@CustomerCode
```

### 5.3.3 Variant 3: use a file with error message texts in language mutations

This variant is the same as the previous one, but we will show the possibility of attaching error message texts. The advantage is that these texts and their language mutations can be provided separately from the program. The method "err" uses the X-Definition reporter, and the report models can be passed to the program as a file with properties (see Java program). In the "err" method, we must, in this case, use a message identifier as the first parameter, the second parameter may be the default message text or an empty string and the third parameter may be the modification string with parameters (if required). If the message identifier is found, the

content of the second parameter is replaced by the text from the found report, otherwise, it will use this parameter to build the report. If we know the message exists, the second parameter may be an empty string or null.

*X-definition „src/task3/Order2a.xdef“:*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Order" root="Order">

<xd:declaration>
<![CDATA[
  /* Information about the customers and products. */
  external Element customers, items;

  /* Validation of customer code. */
  boolean customer() {
    String s = getText();/* get value of the attribute. */
    /* find customer description and save it to Container. */
    Container c = xpath('Customer[@CustomerCode="' + s + '"]', customers);
    /* Check if the customer was found. If yes, return true; otherwise
     * call the method error, which writes the error report and returns false. */
    return c.getLength()==0 ?
      error("POBJ001", "Customer code: &{0}", "&{0}" + s) : true;
  }

  /* Validate product code from Item. */
  boolean item() {
    String s = getText(); /* get attribute value. */
    /* Find product description and save it to Container. */
    Container c = xpath('Product[@Code="' + s + '"]', products);
    /* Check if the code was found. If yes, return true; otherwise
     * call the method error, which writes the error report and returns false. */
    return c.getLength()==0 ?
      error("POBJ002", "Product code: &{0}", "&{0}" + s) : true;
  }
]]>
</xd:declaration>

<Order Number="int" CustomerCode= "customer()">

  <DeliveryPlace>
    <Address Street="string(2,100)"
      House="int(1,9999)"
      City="string(2,100)"
      ZIP="num(5)"/>
  </DeliveryPlace>

  <Item xd:script="occurs 1..10"
    ProductCode="item()"
    Quantity="int(1,1000)"/>
</Order>

</xd:def>
```

Note that in the message texts the "&{#SYS000}" link is added, which allows you to write the rows, the column, the name of the source file, and the XPath position, in the object currently being processed (by the parameters "&{line}", "&{column}", "&{source}" etc). Properties "_prefix" and "_language" are obligatory and specify the prefix of messages and the ISO name of the language in the message file. The file must be recorded in the UTF-8 character code.

*Reports in the Czech language „src/task3/Order_ces.properties“:*

```
# Prefix of messages.
_prefix=POBJ

# ISO name of the language.
_language=ces

# ISO name of the default language.
_defaultLanguage=eng

# ************** Messages: **************
POBJ001=Chybný kód zákazníka: "&{0}"&{#SYS000}
POBJ002=Chybné číslo položky: "&{0}"&{#SYS000}
POBJ003=Chyba ve vstupních datech
POBJ004=Vstupní data jsou zapsána
```

*Reports in the English language „src/task3/Order_eng.properties“:*

```
# Prefix of messages.
_prefix=POBJ

# ISO name of the language.
```

```
_language=eng

# ISO name of the default language.
_defaultLanguage=eng

# *************** Messages: ***************
POBJ001=Incorrect customer code: "&{0}"&{#SYS000}
POBJ002=Invalid item identifier: "&{0}"&{#SYS000}
POBJ003=Input data error
POBJ004=Input data saved
```

In the Java program, first set by the "setProperty(...)" method to access the files with the language message mutations. We can also set the language in which the messages are displayed (if the language is not set, the system setting is used. If the language (or the local language message identifier) is not found in the tables, the default language (i.e. English) is used. If even no such message is found, the default text may be specified by the second parameter of the "error" method). We will show that also the Java program we can use the report system (see the method "Report.error" and "Report.info").

*Program „src/task3/Order2a.java“ :*

```
package task3;

import org.xdef.sys.ArrayReporter;
import org.xdef.sys.Report;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDConstants;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;

public class Otder2a {
        public static void main(String[] args) throws Exception {
                // Compile X-definitions to XDPool
                XDPool xpool = XDFactory.compileXD(null, "src/task3/Order2a.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Order");

                // Set files with reports
                xdoc.setProperty(XDConstants.XDPROPERTY_MESSAGES + "POBJ", "src/task3/*.properties");

                // Set the actual language for reporter  (you can also try to set "ces")
                xdoc.setProperty(XDConstants. XDPROPERTY_MSGLANGUAGE, "eng"); // English

                // Set external variables "products" and "customers"
                xdoc.setVariable("products", "task3/input/Products.xml");
                xdoc.setVariable("customers", "task3/input/Customers.xml");

                // Prepare error reporter
                ArrayReporter reporter = new ArrayReporter();

                // Run the validation mode (you can also try task3/input/Order_err.xml)
                xdoc.xparse("task3/input/Order.xml", reporter);

                // Check errors
                if (reporter.errorWarnings()) {
                        // print errors to the file
                        PrintStream ps = new PrintStream("task3/errors/Order_err.txt");
                        reporter.printReports(ps); //print errors
                        ps.close();
                        // print the message to the system err stream
                        Report rep = Report.error("POBJ003", null);
                        System.err.println(rep.toString());
                } else {
                        // Write processed document to the file
                        KXmlUtils.writeXml("task3/output/Order.xml", xdoc.getElement());
                        // print the message to the system out stream
                        Report rep = Report.info("POBJ004", null);
                        System.out.println(rep.toString());
                }
        }
}
```

In the event of an error, the error message log will contain information with the text in the "error" method (the language is set to "eng" - English):

*The output of the  incorrect item  „task3/errors/Order_err.txt“ in English:*

```
E POBJ001: Incorrect customer code: 'AGFA'; line=2; column=35;
source="file:/D:/cvs/DEV/java/examples/task3/input/Order_err.xml"; xpath=/Order/@CustomerCode; X-
position=Order#Order
```

To solve generating an error file using methods, let the reader try to design one themselves as an exercise.

## 5.4 Task 4

In this task, we will re-validate the data, but add some information to the resulting document. Again, we will work with an input file, the size of which is unlimited. We will add a data check to the solution as in the previous example, but the task will be somewhat more complicated. The customer address element is not part of the input file this time, but we will add it from the data stored in the customer information file. Besides, we'll add item information which we calculate as a multiple of the number of items and the cost per item we get from the item information data. Thus, the attribute "Price" increases in item elements. We only write the correct orders in the output file and the error file. For simplicity, we assume the formal accuracy of the input data (all required data is in the correct format). We leave the error file for simplicity as a text. The input data this time will not contain the "DeliveryPlace" element.

*Input data with orders: „task4/input/Orders.xml“:*

```
<Orders id="123456789">
  <Order Number="123" CustomerCode="ALFA">
    <Item ProductCode="0002" Quantity="2"/>
    <Item ProductCode="0003" Quantity="1"/>
  </Order>
  <Order Number="124" CustomerCode="BETA">
    <Item ProductCode="0001" Quantity="5"/>
  </Order>
</Orders>
```

### 5.4.1 Variant 1: no external Java methods

In the X-definition code, we add the missing element "DeliveryPlace" and add the "Price" attribute to the order items. We will describe the "Order" input data model, and because the output form of an order differs from the input data form, we describe the output order as a separate model that we will use to construct the result. XML data with a list of valid customers and items will be passed as to the respective external variables "products" and "customers" by the program. The program also passes the output channel for writing the result to the variable "output".

*X-defintioní **„src/task4/Orders1.xdef“:***

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Orders" root="Orders">

<xd:declaration>
  external Element products, /* information about products. */
        customers; /* information about customers. */
  external XmlOutStream output; /* Output stream (set by the external program). */

  Container c; /* information about customer created by the method "customer". */
  int errors = 0, errorsOld = 0, errorsOld1 = 0, count = 0; /* counters */

  /* Check customer code */
  boolean customer() {
    String s = getText(); /* get attribute value. */
    c = xpath('Customer[@CustomerCode="' + s + '"]', customers); /* Find the customer. */
        /* Check if the customer found */
    if (c.getLength()==0) {
      /* Customer not found, increase error counter and report an error. */
      errors++;
      return error("Incorrect customer code: " + s); /* sets error message and returns false. */
    }
    return true; /* Customer found, OK */
  }

  /* Check the item code */
  boolean item() {
      String s = getText(); /* get attribute value. */
      /* Find the description of the item. */
      Container c = xpath('Product[@Code="' + s + '"]', products);
      if (c.getLength()==0) { /* Item found? */
        /* Item not found, increase the error counter and report error. */
        errors++;
```

```
        return error("Incorrect item number: " + s);
      }
      return true; /* Item was found, OK */
  }

  /* Write order. */
  void writeObj() {
    if (errors != errorsOld || errorsOld1 != errors()) {
      /* An error occured */
      errorsOld = errors; /* save the counter to "errorsOld"; write nothing. */
      errorsOld1 = errors();
    } else {
        /* Is it the first record? */
        if (count++ == 0) {
                /* nothing was written yet, write XML header and root element. */
                output.setIndenting(true); /* set output indentation. */
                output.writeElementStart(getRootElement());
        }
        /* Create object from the context with the actual order and write it. */
        output.writeElement(xcreate('Order', getElement()));
    }
  }

  /* Close output. */
  void closeAll() {
      /* Something was written?. */
    if (count != 0) {
      /* yes, close output. */
      output.close();
    }
  }

  /* Calculate the total cost of the item of an order. */
  float price() {
    /* Get quantity */
    int number = parseInt(xpath("@Quantity")); /* Get quantity from the input data. */
        String code = from("@ProductCode"); /* Get commodity code from the input data. */
        /* Find the product (we already know that it exists - see customer()). */
        Element el1 = xpath('Product[@Code="' + code + '"]', products).getElement(0);
    float price = parseFloat(el1.getAttr("Price")); /* get price of one product. */
    return price * number; /* Compute total cost and return it. */
  }
  uniqueSet checkObjId int();
</xd:declaration>

<Orders xd:script="finally output.close();" id="string(9)">
  <Order xd:script="occurs +; finally writeObj(); forget"
      Number="checkObjId.ID()"
      CustomerCode="customer()">
    <Item xd:script="occurs 1..100;" ProductCode="item()" Quantity="int"/>
  </Order>
</Orders>

<Order Number="string" CustomerCode="string">
  <DeliveryPlace xd:script="create c" >
    <Address Street="string(2,100)"
      House="int(1,9999)"
      City="string(2,100)"
      ZIP="num(5)"/>
  </DeliveryPlace>
  <Item xd:script="occurs 1..100;"
    ProductCode="string();"
    Quantity="int();"
    Price="float; create price();" />
</Order>

</xd:def>
```

The program to run the task is similar to the previous case.

*Program „src/task4/Orders1.java“ :*

```
package task4;

import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;
import java.util.Properties;

public class Orders1 {
        public static void main(String[] args) throws Exception {
                // compile XDPool from the X-definition
```

```
                Properties props = new Properties();
                XDPool xpool = XDFactory.compileXD(props, "src/task4/Orders1.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Orders");

                // set variables "products", "customers" and "output"
                xdoc.setVariable("products", "task4/input/Products.xml");
                xdoc.setVariable("customers", "task4/input/Customers.xml");
                xdoc.setVariable("output",
                        XDFactory.createXDXmlOutStream("task4/output/Orders.xml", "UTF-8", true));

                // prepare error reporter
                ArrayReporter reporter = new ArrayReporter();

                // run validation mode (you can also try task4/input/Order_err.xml)
                xdoc.setProperties(props);S
                xdoc.xparse("task4/input/Orders.xml", reporter);

                // check errors
                if (reporter.errorWarnings()) {
                        // write log file with errors
                        PrintStream ps = new PrintStream("task4/errors/Orders_err.txt");
                        reporter.printReports(ps); //print errors
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK");
                }
        }
}
```

The output of the program with the added address and price will be stored in the file "task4/output/Orders.xml":

```
<Orders id="123456789">
  <Order CustomerCode="ALFA"
    Number="123">
    <DeliveryPlace>
      <Address City="KLADNO"
        House="5"
        Street="KLADENSKÁ"
        ZIP="54321"/>
    </DeliveryPlace>
    <Item Price="4680.0"
      ProductCode="0002"
      Quantity="2"/>
    <Item Price="3455.0"
      ProductCode="0003"
      Quantity="1"/>
  </Order>
  <Order CustomerCode="BETA"
    Number="124">
    <DeliveryPlace>
      <Address City="MĚLNÍK"
        House="5"
        Street="MĚLNICKÁ"
        ZIP="45321"/>
    </DeliveryPlace>
    <Item Price="602.5"
      ProductCode="0001"
      Quantity="5"/>
  </Order>
</Orders>
```

### 5.4.2 Variant 2: with external Java methods

*X-definition „src/task4/Orders2.xdef“:*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="Orders" root="Orders">

<xd:declaration>
  external method {
    boolean task4.Orders2ext.customer(XXData);
    boolean task4.Orders2ext.item(XXData);
    String task4.Orders2ext.price(XXNode);
    void task4.Orders2ext.closeAll(XXNode);
    void task4.Orders2ext.writeObj(XXNode);
  }
  Element address;
  uniqueSet checkObjId int();
</xd:declaration>
```

```
<Orders xd:script="finally closeAll()" id="num(9)">
  <Order xd:script="occurs +; finally writeObj(); forget"
      Number="checkObjId.ID()"
      CustomerCode="customer()">
    <Item xd:script="occurs 1..100;"
      ProductCode="item()"
      Quantity="int"/>
   </Order>
</Orders>

<Order Number="int" CustomerCode="string">
  <DeliveryPlace>
    <Address xd:script="create address" Street="string(2,100)"
      House="int(1,9999)"
      City="string(2,100)"
      ZIP="num(5)"/>
  </DeliveryPlace>
  <Item xd:script="occurs 1..100;"
    ProductCode="string;"
    Price="float; create price()"
    Quantity="int"/>
</Order>

</xd:def>
```

*Java program „src/task4/Orders2.java“:*

```
package task4;

import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;

public class Orders2 {
        public static void main(String[] args) throws Exception {
                // compile XDPool from the X-definition
                XDPool xpool = XDFactory.compileXD(null, "src/task4/Orders2.xdef");

                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("Orders");

                // set the instance of Orders2ext as user object
                xdoc.setUserObject(new Orders2ext(xpool, // the instance of Orders2ext
                  "task4/input/Products.xml", // file with the infromation obout commodity items
                  "task4/input/Customers.xml", // file with the information abou customers
                  "task4/output/Orders.xml")); // output file.

                // Prepare error reporter
                ArrayReporter reporter = new ArrayReporter();

                // run validation mode (you can also try task4/input/Order_err.xml)
                xdoc.xparse("task4/input/Orders.xml", reporter);

                // Check errors
                if (reporter.errorWarnings()) {
                        // write log file with errors
                        PrintStream ps = new PrintStream("task4/errors/Orders_err.txt");
                        reporter.printReports(ps);
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK");
                }
        }
}
```

*Java source code with external methods „src/task4/Orders2ext.java“:*

```
package task4;

import org.xdef.sys.ArrayReporter;
import org.xdef.xml.KXmlOutStream;
import org.xdef.xml.KXmlUtils;
import org.xdef.XDDocument;
import org.xdef.XDPool;
import org.xdef.proc.XXData;
import org.xdef.proc.XXNode;
import java.io.IOException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;
```

```
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

/** External methods called from Order2.xdef */
public class Orders2ext {

	Element _items, _customers;
	int _errors, _errorsOld, _count;
	KXmlOutStream _output;
	XDPool _xpool;

	private final XPath _xp = XPathFactory.newInstance().newXPath();

	public Orders2ext(XDPool xpool,
		String items,
		String customers,
		String output) throws IOException {
		_items = KXmlUtils.parseXml(items).getDocumentElement();
		_customers = KXmlUtils.parseXml(customers).getDocumentElement();
		_output = new KXmlOutStream(output, "UTF-8", true);
		_xpool = xpool;
		_errorsOld = _errors = _count = 0;
	}

	public static boolean customer(XXData xdata) {
		/* Get the instance of this class. */
		Orders2ext u = (Orders2ext) xdata.getUserObject();
		String s = xdata.getTextValue(); /* get attribute value. */
		try {
			/* Find the customer. */
			XPathExpression xExpr = u._xp.compile("Customer[@CustomerCode='" + s + "']");
			NodeList nl =
				(NodeList) xExpr.evaluate(u._customers, XPathConstants.NODESET);
			if (nl == null || nl.getLength() == 0) {
				/*customer not found, increase erro counter and report an error.*/
				u._errors++;
				xdata.error("Incorrect customer code: " + s, null);
				return false; /* returns false -> incorrect. */
			}
			return true; /* Customer found, OK */
		} catch (Exception ex) {
			u._errors++;
			xdata.error("Unexpected exception: " + ex, null);
			return false;
		}
	}

	public static boolean item(XXData xdata) {
		/* Get the instance of this class. */
		Orders2ext u = (Orders2ext) xdata.getUserObject();
		String s = xdata.getTextValue(); /* get attribute value. */
		try {
			/* Find description of the Item according to code. */
			XPathExpression xExpr = u._xp.compile("Product[@Code='" + s + "']");
			NodeList nl = (NodeList) xExpr.evaluate(u._items, XPathConstants.NODESET);
			if (nl == null || nl.getLength() == 0) {
				/* Item not found, increase the error counter and report an error. */
				u._errors++;
				xdata.error("Incorrect item number: " + s, null);
				return false;
			}
			return true; /* Item was found, OK */
		} catch (Exception ex) {
			u._errors++;
			xdata.error("Unexpected exception: " + ex, null);
			return false;
		}
	}

	public static void writeObj(XXNode xnode) {
		/* Get the instance of this class. */
		Orders2ext u = (Orders2ext) xnode.getUserObject();
		if (u._errors != u._errorsOld) {
			/* an new error occured, do not write recore*/
			u._errorsOld = u._errors; /* save error counter to "errorsOld". */
		} else {
			/* Check if this the first record. */
			if (u._count++ == 0) {
				/* first time, so write the root element. */
				u._output.setIndenting(true); /* nastaveni indentace na vystupu. */
				u._output.writeElementStart(
				xnode.getElement().getOwnerDocument().getDocumentElement());
```

```
                        }
                        /* Prepare XDDocument for the construction according model "Order". */
                        XDDocument xdoc = u._xpool.createXDDocument("Orders");
                        xdoc.setUserObject(u);
                        Element el = xnode.getElement();
                        xdoc.setXDContext(el);
                        try {
                                String s = el.getAttribute("CustomerCode");
                                XPathExpression xExpr = u._xp.compile(
                                        "Customer[@CustomerCode='" + s + "']/Address");
                                NodeList nl = (NodeList) xExpr.evaluate(
                                        u._customers, XPathConstants.NODESET);
                                Element adresa = (Element) nl.item(0);
                                xdoc.setVariable("address", adresa);
                                /* Create the object and write it. */
                                ArrayReporter reporter = new ArrayReporter();
                                u._output.writeNode(xdoc.xcreate("Order", reporter));
                        } catch (Exception ex) {
                                // do nothing, an error was already reported when parsed
                        }
                }
        }

        public static void closeAll(XXNode xnode) {
                /* Get the instance of this class. */
                Orders2ext u = (Orders2ext) xnode.getUserObject();
                /* Check if a record was written. */
                if (u._count != 0) {
                        /* Yes, close the stream. */
                        u._output.writeElementEnd();
                        u._output.closeStream();
                }
        }

        public static String price(XXNode xnode) {
                try {
                        /* Get the instance of this class. */
                        Orders2ext u = (Orders2ext) xnode.getUserObject();
                        Element el = xnode.getXDContext().getElement(); /* Actual context. */
                        String s = el.getAttribute("ProductCode"); /* get commodity code. */
                        /* Find the item description. */
                        XPathExpression xExpr = u._xp.compile("Product[@Code='" + s + "']");
                        /* We already know that it exists. */
                        NodeList nl = (NodeList) xExpr.evaluate(u. items, XPathConstants.NODESET);
                        Element el1 = (Element) nl.item(0);
                        /* get qouantity as a number */
                        int pocet = Integer.parseInt(el.getAttribute("Quantity"));
                        /* compute price */
                        float cena = Float.parseFloat(el1.getAttribute("Price")) * pocet;
                        /* Return it as a string */
                        return String.valueOf(cena);
                } catch (Exception ex) {return "-1"; /* this never should happen! */}
        }
}
```

# 6 JSON/XON data

In this chapter, we discuss some examples with data in JSON/XON format. Our task processes the input data and checks their accuracy. In the examples, we describe again both the source text of the Java program as well as the X-definitions and input data and are available in the functional form in the files that you can download this together with the X-definition Jar file at http://www.xdefinice.cz/en/download.
Note that because JSON format is a subset of XON format, the description of the model or the processed data may be either in the JSON or in the XON format.

## 6.1 Task5 (JSON)

Let's have following JSON data *(see „task5/input/JsonExample.json"):*

```
{ "date" : "2020-02-22",
  "cities" : [
    { "from": ["Brussels",
        {"to": "London", "distance": 322},
        {"to": "Paris", "distance": 265}
      ]
    },
    { "from": ["London",
```

```
        {"to": "Brussels", "distance": 322},
        {"to": "Paris", "distance": 344}
      ]
    }
  ]
}
```

The model of this JSON data is written in the element xd:json with name "distancies". The complete X-definition (see *„src/task5/ JsonExample.xdef“):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="JsonExample" root="distancies">
  <xd:json xd:name="distancies">
    { "date" : "date()",
      "cities" : [
        {$script = "occurs +",
          "from": [
            "string()",
            {$script = "occurs +",
               "to":       "jstring()",
               "distance": "int(0, 9999)"
            }
          ]
        }
      ]
    }
  </xd:json>
</xd:def>
```

Note that the model of JSON data contains a JSON text where the description of values is a string with X-script description. If you want to add an X-script command to maps or arrays you write it to the item $sctipt = "text of X-script command".

In the Java program for parsing JSON data, you use the method „jparse“ instead of “xparse”. The result of parsing is an XON object. You can print this object in JSON format using the static method “toJsonString” from the class “org.xdef.xon.XonUtil”.

The complete Java program (see *„src/task5/JsonExample.java“):*

```
package task5;

import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;
import java.io.Writer;
import java.util.Properties;
import org.xdef.xon.XonUtil;

public class JsonExample {
        public static void main(String... args) throws Exception {
                // compile XDPool from the X-definition
                Properties props = new Properties();
                XDPool xpool = XDFactory.compileXD(props, "src/task5/JsonExample.xdef");
                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("JsonExample");
                // prepare error reporter
                ArrayReporter reporter = new ArrayReporter();
                xdoc.setProperties(props);
                Object xon = xdoc.jparse("task5/input/JsonExample.json", reporter);
                // check errors
                if (reporter.errorWarnings()) {
                        // write log file with errors
                        PrintStream ps = new PrintStream("task5/errors/distancies_err.txt");
                        reporter.printReports(ps); //print errors
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK. See task5/output/JsonExample.json");
                        // Store the parsed result
                        Writer out = new OutputStreamWriter(
                                new FileOutputStream("task5/output/JsonExample.json"), "UTF-8");
                        out.write(XonUtil.toJsonString(xon, true));
                        out.close();
                }
        } }
```

## 6.2 Task6 (XON)

Let's have following XON data *(see „task6/input/XonExample.xon"):*

```
{ date = D2020-02-22,
  cities = [
    {from = [
        "Brussels",
        { to = "London", distance = 322I},
        { to = "Paris", distance = 265I}
      ]
    },
    {from = [
        "London",
        { to = "Brussels", distance = 322I},
        { to = "Paris", distance = 344I}
      ]
    }
  ]
}
```

The model of this JSON data is written in the element xd:json with the name "distancies". Note that the model is written to xd:json element (in fact you can always use JSON or XON format ant it is equivalent).

The complete X-definition (see *„src/task6/XonExample.xdef"):*

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.1" name="XonExample" root="distancies">
  <xd:json xd:name="distancies">
    { date = "date()",
      cities = [
        {$script = "occurs +",
          from = [
            "string()",
            {$script = "occurs +",
               to =       "jstring()",
               distance = "int(0, 9999)"
            }
          ]
        }
      ]
    }
  </xd:json>
</xd:def>
```

Note that the model of XON data again contains a JSON text where the description of values is a string with X-script description. The description of an XON model can be either in the JSON format or in the XON format.

In the Java program for parsing JSON data, you use the method „jparse" instead of "xparse". The result of parsing is an XON object. You can print this object in JSON format using the static method "toXonString" from the class "org.xdef.xon.XonUtil".

The complete Java program (see *„src/task6/XonExample.java"):*

```
package task6;

import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import org.xdef.sys.ArrayReporter;
import org.xdef.XDDocument;
import org.xdef.XDFactory;
import org.xdef.XDPool;
import java.io.PrintStream;
import java.io.Writer;
import java.util.Properties;
import org.xdef.xon.XonUtil;

public class XonExample {
        public static void main(String... args) throws Exception {
                // compile XDPool from the X-definition
                Properties props = new Properties();
                XDPool xpool = XDFactory.compileXD(props, "src/task6/XonExample.xdef");
                // Create the instance of XDDocument object (from XDPool)
                XDDocument xdoc = xpool.createXDDocument("XonExample");
                // prepare error reporter
                ArrayReporter reporter = new ArrayReporter();
                xdoc.setProperties(props);
                Object xon = xdoc.jparse("task6/input/XonExample.xon", reporter);
                // check errors
                if (reporter.errorWarnings()) {
                        // write log file with errors
```

```
                        PrintStream ps = new PrintStream("task6/errors/distancies_err.txt");
                        reporter.printReports(ps); //print errors
                        ps.close();
                        System.err.println("Incorrect input data");
                } else {
                        System.out.println("OK. See task6/output/XonExample.xon");
                        // Store the parsed result
                        Writer out = new OutputStreamWriter(
                                new FileOutputStream("task6/output/XonExample.xon"), "UTF-8");
                        out.write(XonUtil.toXonString(xon, true));
                        out.close();
                }
        }
}
```