



# Xcomponents 4.2

## *User Manual*

Author:	Jindřich Kocman, Václav Trojan
Version:	4.2.0.0
Date:	2022-06-15



# Contents

<b>Xcomponents 4.2 .....</b>	<b>1</b>
<b>1 Notice.....</b>	<b>1</b>
<b>2 Introduction.....</b>	<b>1</b>
2.1 Terms and abbreviations .....	1
<b>3 Xcomponents.....</b>	<b>1</b>
3.1 An instance of an Xcomponent.....	3
3.2 XDPosition.....	3
3.3 Commands to generate Xcomponents .....	3
3.3.1 <i>Command %class</i> .....	3
3.3.2 <i>Command %bind</i> .....	4
3.3.3 <i>Command %interface</i> .....	5
3.3.4 <i>Command %ref</i> .....	5
3.3.5 <i>Command %enum</i> .....	6
<b>4 Xcomponents created from JSON and other data models .....</b>	<b>7</b>
4.1 Xcomponents created from JSON map.....	7
4.2 Xcomponents created from JSON array.....	7
<b>5 How to work with the Xcomponents.....</b>	<b>9</b>
5.1 Generation of Xcomponents.....	9
5.2 Create an instance of Xcomponent from XML data (unmarshalling) .....	9
5.3 Create XML from Xcomponent (marshaling) .....	9
5.4 Transformation to another Xcomponent.....	9
<b>6 Create Xcomponents and work with them .....</b>	<b>10</b>
6.1 Example of transformation .....	11

## Inventory of changes

Default version Xdefinition 4.2, build 4.2.0.5



# 1 Notice

Questions, remarks, and bug reports please send to: [xdef@syntea.cz](mailto:xdef@syntea.cz).

The actual version of Xdefinition can be downloaded from <https://github.com/Syntea/xdef> or <http://www.xdefinice.cz/en/>

## 2 Introduction

This document is a user's guide that describes the basic use of the "Xcomponents" technology. It is designed for programmers who need to convert data from XML into Java objects and back. Xcomponents is built on top of the Xdefinition and extends its capability of working with XML data. For using Xcomponents, a basic knowledge of the Xdefinition is required.

### 2.1 Terms and abbreviations

<b>Xdefinition</b>	1. The language used for the description of the structure, content, processing, and construction of XML objects. 2. XML element in the Xdefinition language.
<b>Xcomponents</b>	Technology for creating Java classes from the Xdefinitions
<b>Xcomponent</b>	Java class created by the Xcomponent technology.
<b>XComponent</b>	Java interface org.xdef.component.XComponent implemented by Xcomponents.
<b>Unmarshalling</b>	Populate an object with data from an XML source.
<b>Marshalling</b>	Data conversion of an object into XML. Opposite of unmarshalling.
<b>xd:component</b>	An XML element containing a list of descriptions of Xcomponents in the Xdefinition
<b>XDPosition</b>	A description of a location of a model in the set of Xdefinitions (compiled into XDPool)

## 3 Xcomponents

The Xcomponent is a Java source code of a class generated according to a particular in an Xdefinition model. Each model of an element in the Xdefinition corresponds to one generated Xcomponent. The values of attributes, elements, and text values in the Xcomponent, are accessible by the methods **getNAME** and **setNAME**, where "**NAME**" is the name of the attribute or an element in the Xdefinition. The conversion of the value of an attribute or a text node described in the Xdefinition and the type of object in Java is given by the preset relations shown in the following table:

Xdefinition datatype	Xcomponent
anyURI	java.net.URI
Boolean	java.lang.Boolean
byte, short, int, long, integer, float, double, and derived types	java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.math.BigInteger
Char	Java.lang.Character
float, double	java.lang.Float, java.lang.Double
decimal, dec	java.math.BigDecimal
datetime, xdatetime, gDate, gTime, gYear, gMoth, gYearMonth, gMonthDay	org.xdef.sys.SDatetime (also available as java.util.Calendar, java.util.Date, java.sql.Timestamp)
Duration	org.xdef.sys.SDuration
base64Binary, hexBinary, base64, hex	byte[]
Enum	Java.lang.String Or Enum in Java
Currency	java.util.Currency
emailAddr	org.xdef.XDEmailAddr
Gps	org.xdef.sys.GPSPosition
ipAddr	java.net.InetAddress

Price	org.xdef.sys.Price
Telephone	org.xdef.XDTelephone
-----	
other datatypes	java.lang.String

The values of child elements are represented by Xcomponent objects. If the maximum number of the quantifier is higher than one then it is represented by `java.util.List<correspondent Xcomponent>`. Also if more than one text value occurs at a position it will be represented as `java.util.List<correspondent datatype>`.

All types are objects (not primitive values), so it is possible to test in a program if a value exists in the Xcomponent object (e.g. optional values may not exist). If an element or a value have more occurrences, it creates an array of values (using `java.util.List`), and instead of a setter, the method `addNAME` is generated. Because there are more possibilities to work with datatypes of `DateTime` the getters are generated in more variants: `timestampOfNAME`, `calendarOfNAME`, and `dateOfNAME`.

*Note some of the commands in the Xdefinition affect the creation of the Xcomponent*

- *ignore* – the Xdefinition object is ignored and its code is not generated in the Xcomponent.
- *forget* – the elements are deleted from the memory after processing the document. However, the corresponding Xcomponent is created

### Example

Let's have a master Xdefinition describing an insurance contract:

```
<Contract
  VIN   = "required int()"
  Date  = "required date()"
  <Owner
    Name      = "required string()"
    CompanyID = "required num(8)" />
  <Keeper xd:script = "occurs 1.."
    PersonalID = "required num(10)" />
</Contract>
```

Generated Xcomponent will look as follows (the constructors and the methods of the interface `org.xdef.component.XComponent` are not displayed):

```
package user.kocman.manual;

public class Contract implements org.xdef.component.XComponent{
  public org.xdef.sys.SDatetime getDate() {return _Date;}
  public java.util.Date dateOfDate() {
    return org.xdef.sys.SDatetime.getDate(_Date);
  }
  public java.sql.Timestamp timestampOfDate() {
    return org.xdef.sys.SDatetime.getTimestamp(_Date);
  }
  public java.util.Calendar calendarOfDate() {
    return org.xdef.sys.SDatetime.getCalendar(_Date);
  }
  public Long getVIN() {return _VIN;}
  public Contract.Owner getOwner() {return _Owner;}
  public java.util.List<Contract.Keeper> listOfKeeper() {return _Keeper;}
  public void setDate(org.xdef.sys.SDatetime x) {_Date = x;}
  public void setDate(java.util.Date x) {
    _Date = x==null ? null : new org.xdef.sys.SDatetime(x);
  }
  public void setDate(java.sql.Timestamp x) {
    _Date = x==null ? null : new org.xdef.sys.SDatetime(x);
  }
  public void setDate(java.util.Calendar x) {
    _Date = x==null ? null : new org.xdef.sys.SDatetime(x);
  }
  public void setVIN(Long x) {_VIN = x;}
  public void setOwner(Contract.Owner x) {_Owner = x;}
  public void addKeeper(Contract.Keeper x) {
    if (x!=null) _Keeper.add(x);
  }
  private org.xdef.sys.SDatetime _Date;
  private Long _VIN;
  private Contract.Owner _Owner;
  private final java.util.List<Contract.Keeper> _Keeper =
    new java.util.ArrayList<Contract.Keeper>();
}
```

```
// Constructors and implementation of the interface XComponent ...

public static class Owner implements org.xdef.component.XComponent{
    public String getICO() {return _ICO;}
    public String getName() {return _Name;}
    public void setICO(String x) {_ICO = x;}
    public void setName(String x) {_Name = x;}
    public String xposOfICO(){return XD_XPos + "/@ICO";}
    public String xposOfName(){return XD_XPos + "/@Name";}
    // Constructors and implementation of the interface XComponent ...
}

public static class Keeper implements org.xdef.component.XComponent{
    public String getPersonalID() {return _PersonalID;}
    public void setPersonalID (String x) {_PersonalID = x;}
    public String xposOfRC(){return XD_XPos + "/@PersonalID ";}
    private String _PersonalID;
    // Constructors and implementation of the interface XComponent ...
}
}
```

### 3.1 An instance of an Xcomponent

Each Xcomponent implements Java interface `org.xdef.component.XComponent` that allows conversion from/to the XML data format. When it comes to populating the appropriate data from the XML (operation unmarshalling), AN Xcomponent instance can be created when parsing the XML according to the model in the Xdefinition. Xcomponent can also be created using the constructor and populated with the data by the Java program. From an instance of an Xcomponent, you can create the XML element by calling the method `org.xdef.XComponent.toXml()` (marshaling operation).

### 3.2 XDPosition

XDPosition is a description of a location of a model in the set of Xdefinitions (compiled to XDPool object). It consists of the name of Xdefinition, followed by the "#" and the name of the model. A model can be followed by the "/" character, and either the name of the nested element "\$text" for a text value, or "@", and the name of the attribute. If the name contains a prefix of namespace it is written with the prefix, as listed in the Xdefinition. If a set of descendants of an element contains more nodes with the same name, it is described by the serial number of this item in square brackets (the numbers start from one: "[1]"). The first element is taken if the number is not specified (i.e. "[1]" doesn't have to be specified to be written).

In the following example are various XDPositions of individual items:

```
<xd:def name = "Model">
  <A>                                Model#A
    <B>                                Model#A/B
      b = "string()" />              Model#A/B/@b
    <C />                             Model#A/C
    <B>                                Model#A/B[2]
      required string();             Model#A/B/$text
    </B>
  </A>
</xd:def>
```

### 3.3 Commands to generate Xcomponents

The list of commands used for creating an Xcomponent is written in the Xdefinition as the text of the element `<xd:component>`. The `<xd:component>` element may be written in a separate Xdefinition or it can be part of any other Xdefinition. Each command in the list is terminated with ";".

#### 3.3.1 Command %class

The command `%class` defines a Java class generated according to a model of an element in the Xdefinition. The `%class` keyword is followed by a fully qualified class name and by the keyword `%link` which specifies the XDPosition of the model in a set of Xdefinitions, from which the Xcomponent is generated. If the Xcomponent extends a Java class or it implements a Java interface then you can specify "extends SuperClassName

implements InterfaceName" after the name of the Java class, where SuperClassName and InterfaceName must be fully-qualified names. The syntax is the same as in the declaration of a class in the Java language.

### Example 1

Source Xdefinition:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name="Vehicle"
  xd:root="Vehicle">

  <Vehicle
    VIN = "required string()"
  />

  <xd:component>
    %class cz.syntea.tutorial.Vehicle
      extends cz.syntea.tutorial.PreVehicle
      implements cz.syntea.tutorial.IVehicle
      %link Vehicle#Vehicle;
  </xd:component>
</xd:def>
```

Generated Xcomponent:

```
package cz.syntea.tutorial;

public class Vehicle extends cz.syntea.tutorial.PreVehicle
implements cz.syntea.tutorial.IVehicle, org.xdef.component.XComponent{
  public String getVIN() {return _VIN;}
  public void setVIN(String x) {_VIN = x;}
  // Constructors and implementation of the interface XComponent ...
}
```

### 3.3.2 Command %bind

Command %bind can set a new name of an item in an Xdefinition (the attribute model, element model, or text node model and the corresponding names of the getters and setters). The keyword %bind is followed by the name that will be applied instead of the automatically generated one. After the specification of the name, the command continues with the keyword "%with" which is followed by the list of XDPositions (separated by a comma) to which the statement relates. The same name can be used in many models. The getters and setters will be automatically adjusted to match the newly assigned name (see Example 2). If the generated Java class has an ancestor, you can use the %bind command to bind the getter and setter defined in the ancestor. In this case, a given variable, including getters and setters will not be generated and it will use the implementation of these methods in the ancestor.

### Example 2

Let's have a truck that extends the vehicle defined in the previous example. The generated Xcomponent does not contain getters and setters of the items which are inherited:

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name="Truck"
  xd:root="Truck">

  <Truck xd:script = "ref Vehicle#Vehicle"
    MaxWeight = "required int()"
  />

  <xd:component>
    %bind VIN %with cz.syntea.tutorial.Vehicle %link Truck#Truck/@VIN;
    %class cz.syntea.tutorial.Truck
      extends cz.syntea.tutorial.Vehicle
      %link Truck#Truck;
  </xd:component>
</xd:def>
```

Generated Xcomponent:

```
package cz.syntea.tutorial;
public class Truck extends cz.syntea.tutorial.Vehicle
implements org.xdef.component.XComponent{
  public Long getMaxWeight() {return _MaxWeight;}
```



```

public void setMaxWeight(Long x) {_MaxWeight = x;}
private Long _MaxWeight; getVIN()
// The methods getVIN and setVIN are already implemented in the superclass Vehicle!

// Constructors and implementation of the interface XComponent ...
}

```

### 3.3.3 Command %interface

The command `%interface` is used when the (final) model takes the structure of another (referenced) model and, where appropriate, it also adds additional attributes, text values, or elements. To make the Xcomponents generated from the models behave like the Xcomponent created from the referenced ones, you can create an interface from the given model. This interface can be added for generating final models (see 3.3.1). The interface command starts with the keyword `%interface`, which is followed by the fully qualified name of the interface and by the keyword `%link` followed by an XDPosition of the model in the project.

#### Example 3

The personal car shares part of the structure of the Vehicle. This structure was pulled out and the element Personal links to it.

```

<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name="Personal"
  xd:root="Personal">

  <Vehicle
    VIN = "required string()" />

  <Personal xd:script = "ref Vehicle"
    MaxPersons = "required int()" />

  <xd:component>
    %interface cz.syntea.tutorial.IVehicle %link Personal#Vehicle;
    %class cz.syntea.tutorial.Personal
      implements cz.syntea.tutorial.IVehicle
      %link Personal#Personal;
  </xd:component>

</xd:def>

```

The generated interface declares all the getters and setters of the Vehicle class:

```

package cz.syntea.tutorial;

public interface IVehicle extends org.xdef.component.XComponent {
  public String getVIN();
  public void setVIN(String x);
  public String xposOfVIN();
}

```

The Xcomponent Personal implements the interface IVehicle:

```

package cz.syntea.tutorial;

public class Personal implements cz.syntea.tutorial.IVehicle, org.xdef.component.XComponent{
  public Long getMaxPersons() {return _MaxPersons;}
  public String getVIN() {return _VIN;}
  public void setMaxPersons(Long x) {_MaxPersons = x;}
  public void setVIN(String x) {_VIN = x;}
  // Constructors and implementation of the interface XComponent ...
}

```

### 3.3.4 Command %ref

It often happens that the project (XDPool) is generated from many Xdefinitions. In this case, the Xcomponent is generated from a given Xdefinition, but XDPool is different (for example, there are some Xdefinition extra, missing, etc.), it can be used in the already created Xcomponent and it's necessary to avoid its new generation (for example, if the Xcomponent is located in another Jar file). The reference to the already generated Xcomponent is provided by the command `%ref` with the fully qualified name of the already generated Xcomponent and with the keyword `%link` with an XDPosition of the model in XDPool.

#### Example

In the Jar file, which is in the current classpath, there is already the generated Xcomponent **cz.syntea.test.prexd.Vehicle** from Example 1. So, when you create a set of new Xcomponents, the already generated Xcomponent is used and it is not generated again.

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name="Fleet"
  xd:root="Fleet">

  <Fleet>
    <Personal xd:script = "ref Vehicle#Vehicle; occurs 0.."/>
    <Van xd:script = "ref Vehicle#Vehicle; occurs 0.."/>
  </Fleet>

  <xd:component>
    %ref cz.syntea.tutorial.Vehicle %link Vehicle#Vehicle;
    %class cz.syntea.tutorial.Fleet %link Fleet#Fleet;
  </xd:component>
</xd:def>
```

```
package cz.syntea.tutorial;

public class Fleet implements org.xdef.component.XComponent{
  public java.util.List<Vehicle> listOfPersonal () {return _Personal;}
  public java.util.List<Vehicle> listOfVan() {return _Van;}
  public void addPersonal (Vehicle x) {
    if (x != null) {
      if (x.getXPos() == null)
        x.XInit(this, "Personal", null, "Fleet#Fleet/Personal");
      _Personal.add(x);
    }
  }
  public void addVan(Vehicle x) {
    if (x != null) {
      if (x.getXPos() == null)
        x.XInit(this, "Van", null, "Fleet#Fleet/Van");
      _Van.add(x);
    }
  }
  // Constructors and implementation of the interface XComponent ...
}
```

### 3.3.5 Command %enum

If the Xdefinition data type enum is specified, its value in the Xcomponent is represented by default as a String. However, in case we want to have a choice in the code only from a set of allowed values, it is possible to generate the data value of the enum as Java enum type. The data type must be defined in the Script section **<xd:declaration>**. The enum will be generated by using the command **%enum** followed by the fully qualified name of the enum class and the name of the data type.

#### Example

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2" xd:name="Vehicle2" xd:root="Vehicle2">

  <Vehicle2
    VIN = "required string()"
    Color = "required color()"
  />

  <xd:declaration>
    type color enum('white', 'blue', 'green');
  </xd:declaration>

  <xd:component>
    %enum cz.syntea.tutorial.Color color;
    %class cz.syntea.tutorial.Vehicle2 %link Vehicle2#Vehicle2;
  </xd:component>
</xd:def>
```

```
package cz.syntea.tutorial;
public enum Color implements org.xdef.component.XCEnumeration {
  white,
  blue,
  green;
}
```

```

@Override
public final Object itemValue() {return name();}
@Override
public final String toString() {return name();}
public static final Color toEnum(final Object x) {
    if (x != null)
        for(Color y: values())
            if (y.itemValue().toString().equals(x.toString())) return y;
    return null;
}
}

```

```

package cz.syntea.tutorial;

public class Vehicle2 implements org.xdef.component.XComponent{
    public cz.syntea.tutorial.Color getColor() {return _Color;}
    public String getVIN() {return _VIN;}
    public void setColor(cz.syntea.tutorial.Color x) {_Color = x;}
    public void setVIN(String x) {_VIN = x;}
    // Constructors and implementation of the interface XComponent ...
}

```

## 4 Xcomponents created from JSON and other data models

Xcomponents can be generated not only from XLM element models, but also from JSON/ XON, INI, or CSV models. From an instance of these models, the corresponding XON object can be retrieved using the toXon() method.

### 4.1 Xcomponents created from JSON map

Example of the model of JSON map with named items „a“ and „b“:

```

<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root="A">
<xd:json name="A">
  { a: "int();", b: "date();" }
</xd:json>
<xd:component> %class mytests.MyTestXonA %link #A; </xd:component>
</xd:def>

```

Getters and setters of named values in JSON map have the "\$" character before the name.

The resulting object is a map, containing the values "a" and "b". In the Xcomponent of MyTestXonA, this object is accessed by the getMap\$() method. the getMap\$ method will be generated. The values "a" and "g" are accessed by the corresponding getters and setters:

```

package mytests;

public class MyTestXonA implements org.xdef.component.XComponent {
    ...
    public java.util.Map<String, Object> getMap$() ...

    public Integer get$a() ...
    public org.xdef.sys.SDatetime get$b() ...

    public void set$a(Integer x) ...
    public void set$b(org.xdef.sys.SDatetime x) ...
    ...
}

```

### 4.2 Xcomponents created from JSON array

The names of getters and setters of items of JSON array strts with „item\$“. If more items are described in the array model, then the "\_" character and the item serial number are added after "Item\$". E.g. getItem\$\_1, setItem\$\_2, etc.

Example of model of JSON array with 3 items:

```

<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root="B">
<xd:json name="B">

```

```
[ "int();", "date();", "* boolean();" ]  
</xd:json>  
<xd:component> %class mytests.MyTestXonB %link #B; </xd:component>  
</xd:def>
```

The object in the model „B“ in the Xdefiniton is now an array of values:

```
<xd:def xmlns:xd='http://www.xdef.org/xdef/4.2' root="B">  
<xd:json name="B">  
  [ "int();", "date();", "* boolean();" ]  
</xd:json>  
<xd:component> %class mytests.MyTestXonB %link #B; </xd:component>  
</xd:def>
```

The JSON model of the array can be retrieved using the `getArray$()` method. The individual elements described in the model are accessed using the `getItem$`, `getItem$_1`, `getItem$_2` getters and the `setItem$`, `setItem$_1`, `setItem$_2` setters:

```
package mytests;  
public class MyTestXonB implements org.xdef.component.XComponent {  
  ...  
  public java.util.List<Object> getArray$() ...  
  
  public Integer getItem$() ...  
  public org.xdef.sys.SDatetime getItem$_1() ...  
  public Boolean getItem$_2() ...  
  
  public void setItem$(Integer x) ...  
  public void setItem$_1(org.xdef.sys.SDatetime x) ...  
  public void setItem$_2(Boolean x) ...  
  ...  
}
```

## 5 How to work with the Xcomponents

### 5.1 Generation of Xcomponents

To generate Xcomponents from Xdefinitions you need the object `org.xdef.XDPool` (you can create it for example, by the method `org.xdef.XDFactory.compileXD` or you can load the compiled `XDPool` from a file by the method `org.xdef.XDFactory.readXDPool`, etc.). The Xcomponents are generated using the method `org.xdef.component.GenXComponent.genXComponent()`:

```
XDPool xdPool = XDFactory.compileXD(null, "resources/manual/Vehicle.xdef");
GenXComponent.genXComponent(
    xdPool,                                // XDPool
    new File("src/main/java").getAbsolutePath(), // directory where to generate
    null                                   // character set (default UTF-8)
);
```

The code above generates the source code of Xcomponents defined in the passed `XDPool`. After the compilation of the generated Java classes, they are ready for use.

### 5.2 Create an instance of Xcomponent from XML data (unmarshalling)

`XDPool` is a fully reentrant object, it can be saved as a static final variable that is statically initiated, and therefore it is accessible to all programs that work with it.

Example of creating an Xcomponent from XML:

```
public static final XDPool XD_POOL = XDFactory.compileXD(null, "/manual/Vehicle.xdef");
...

ArrayReporter reporter = new ArrayReporter();
File source = new File("src/main/resources/manual/Vehicle.xml");
...

// 1. Create XDDocument
XDDocument doc = XD_POOL.createXDDocument("Vehicle");
// 2. Create an Xcomponent with the method parseXComponent().
XComponent xc = doc.parseXComponent(source, Vehicle.class, reporter);
// 3. The method parseXComponent returns the Xcomponent created according to the command %class.
//    We can cast it therefore to the class defined by the %class command and use
//    the getters and setters: getMyObjekt(), ... setMyObjekt(...) ...
Vehicle vehicle = (Vehicle) xc;
```

### 5.3 Create XML from Xcomponent (marshaling)

From an Xcomponent, it is possible to create an XML element that corresponds to the values in the Xcomponent. To do this, use the method `toXml()`:

```
Element el = xc.toXml();
```

### 5.4 Transformation to another Xcomponent

Xcomponents allow you to create a new Xcomponent with a different structure (a variant of the construction mode) from a particular Xcomponent.

#### Example

```
XComponent oldxc;                                // original Xcomponent
...
XDPool xp;                                       // given XDPool
XComponent newxc = XComponentUtil.toXComponent(oldc, // new Xcomponent
    xp,                                           // XDPool
    "def#model");                               // XDPosition of the model in the new Xcomponent
```

## 6 Create Xcomponents and work with them

Let us have Xdefinition, which describes a town in which there are streets and houses with tenants (the file `src/data/Town.xdef`):

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name="Town"
  xd:root="Town " >

  <Town Name = "required string()">
    <Street xd:script="occurs 0..;"
      Name = "required string()" >
      <House xd:script="occurs 0..; ref House" />
    </Street>
  </Town>

  <House Num      = "required int()"
    Address = "optional string()">
    <Person xd:script="occurs 0..; ref Person" />
  </House>

  <Person FirstName = "required string()"
    LastName = "required string()" />

</xd:def>
```

XML data are in the file `"src/data/Town.xml"` (processed by Xdefinition "Town"):

```
<Town Name="Nonehill">
  <Street Name="Long">
    <House Num="1">
      <Person FirstName="John" LastName="Smith"></Person>
      <Person FirstName="Jane" LastName="Smith"></Person>
    </House>
    <House Num="2"/>
    <House Num="3">
      <Person FirstName="James" LastName="Smith"></Person>
    </House>
  </Street>
  <Street Name="Short">
    <House Num="1">
      <Person FirstName="Jeremy" LastName="Smith"></Person>
    </House>
  </Street>
</Town>
```

The Xdefinition in the XML file `"src/Town_XC.xdef"` describes the generation of Xcomponents and the interface "Citizen":

```
<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name="Town-XC">

  <xd:component>
    /*****
     *   Xcomponents generated from the Xdefinition Town *
     *****/
    %class town.Town %link Town#Town;
    %class town.House %link Town#House;

    /* Create the interface according to the Xcomponent from Town#Person */
    %interface town.Citizen %link Town#Person;
    /* The Xcomponent has the specified interface */
    %class town.Tenant implements town.Citizen %link Town#Person;
  </xd:component>
</xd:def>
```

Let's show how to create a file in which the XDPool with the compiled Xdefinitions will be saved. Then we'll generate the Xcomponents from it:

```
public class Example_XC {
  public static void main(String[] args) throws Exception {
    // 1. Compile Xdefinitions
```

```

        XDPool xdPool = XDFactory.compileXD(null,          // null -> use System properties as default
            "src/data/*.xdef");                          // Xdefinitions

        // 2. save XDPool to the file
        xdPool.writeXDPool(new File("resources/XDPool.dat"));

        // 3. generate Xcomponents
        GenXComponent.genXComponent(xdPool,              // XDPool
            new File("src").getAbsolutePath(),          // directory where to generate
            null);                                       // character set (default UTF-8)
    }
}

```

Now we can write the program that will use the Xcomponent. We first set the static variable XP (using the method `getXDPool`). From the input file "resources/manual/Town.xml" we create the instance of Xcomponent Town and then we print its contents. Finally, we add the addresses of houses in the Xcomponent and write it to the file `resources/Town_processed.xml`:

```

public class Example_XC1 {
    // read compiled XDPool from the file to XP
    public static final XDPool XP;
    static {
        try {
            XD = XDFactory.readXDPool("resources/XDPool.dat");
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) throws Exception {
        // 1. create XDDocument
        XDDocument xd = XD.createXDDocument("Town");

        // 2. create an instance of Xcomponent Town (unmarshal) from XML data
        File townFile = new File("src/Example_XC1.xml");
        Town town = (Town) xd.parseXComponent(townFile, town.Town.class, null);

        // 3. Print contents of the Xcomponent Town
        System.out.println("Town " + town.getName());
        for (Town.Street street: town.listOfStreet()) {
            System.out.println("Street " + street.getName() + ":");
            for (House house: street.listOfHouse()) {
                System.out.print("House " + house.getNum() + ". ");
                if (dum.listOfPerson().size() > 0) {
                    System.out.println("Tenants :");
                    for (Person person: house.listOfPerson()){
                        System.out.println(person.getFirstName() + " " + person.getLastName());
                    }
                } else {
                    System.out.println("No tenants in this house");
                }
            }
        }

        // 4. Add to each house address.
        for (Town.Street street: town.listOfStreet()) {
            for (House house: street.listOfHouse()) {
                house.setAddress(town.getName() + ", " + street.getName() + " " + house.getNum());
            }
        }

        // 5. Save the XML document with addresses to the file Town_processed.xml (marshall)
        Element el = town.toXml();
        KXmlUtils.writeXml("resources/Town_processed.xml", el, true, false);
    }
}

```

## 6.1 Example of transformation

Let's try to define an Xdefinition describing another XML data containing a list of tenants in the city and let us use the clause `create` to describe how to create a new transformed file (according to the Xdefinition "Tenants ") from the input data:

```

<xd:def xmlns:xd="http://www.xdef.org/xdef/4.2"
  xd:name = "Tenants"
  xd:root = "Tenants">

  <Tenants>
    <Tenant xd:script="occurs 0..; create from('//Person');"
      GivenName = "required string(); create from('@FirstName')"
      FamilyName = "required string(); create from('@LastName')"
      Address = "required string(); create from('../@Address')" />
    </Tenant>
  </Tenants>

  <xd:component>
    /*****
    * Xcomponents of the Xdefinition Tenants *
    *****/
    %class cz.syntea.tutorial.Tenants %link Tenants#Tenants;

    %bind FirstName %link Tenants#Tenants/Tenant/@GivenName;
    %bind LastName %link Tenants#Tenants/Tenant/@FamilyName;
  </xd:component>

</xd:def>

```

Transformation of the Xcomponent Town according to the model Tenants by the method  
**org.xdef.component.ComponentUtil.toXComponent(...):**

```

public class Example_XC2 {

  public static void main(String[] args) throws Exception {
    // 1. create XDPool a XDDocument
    XDDocument xd = Example_XC1.XD.createXDDocument("Town");

    // 2. create the instance of the Xcomponent Town (unmarshal)
    File townFile = new File("resources/Town_processed.xml");
    Town town = (Town) xd.parseXComponent(townFile, Town.class, null);

    // 3. create transformation to the Xcomponent Tenants
    Tenants tenants =
      (Tenants) XComponentUtil.toXComponent(town, Example1.XD, "Residents#Residents");

    // 4. save data to the file Residents.xml
    Element e1 = tenants.toXml();
    KXmlUtils.writeXml("resources/Residents.xml", e1, true, false);

    // 5. print the list of tenants
    for (Tenants.Resident x: tenants.listOfResident()) {
      System.out.println(x.getFirstName() + " " + x.getLastName() + "; " + x.getAddress());
    }
  }
}

```

The source code of the examples above is distributed together with the Xdefinition files and can be downloaded at: <http://www.xdefinice.cz/en/download/> in the folder “examples”.