

XML Enhancements to Java™ (XJ)

User Manual for Release Version 1.1.0

xj@watson.ibm.com



Contents

1	Introduction	3
1.1	Document Conventions	3
1.2	Online Information	3
1.3	Reporting Bugs	3
2	Getting Started	3
2.1	Download	4
2.2	Requirements	4
2.3	Installation	4
2.4	Compiling an XJ Program	4
2.5	Running an XJ program	5
3	XJ Language Overview	5
3.1	XJ Type References	6
3.1.1	Referring to XML Schemas	6
3.1.2	Referring to Element Declarations	6
3.1.3	Schema Built-In Types	7
3.2	Import of XML types	7
3.2.1	Type-on-demand Import of an XML Schema	7
3.2.2	Single-type-import of XML Schema Declarations	7
3.2.3	Default Imports	8
3.3	XJ Generics	8
3.4	Automatic Unboxing	8
3.5	XPath Expressions	8
3.5.1	Referring to In-Scope Variables	9
3.5.2	Static Semantics	9
3.5.3	Runtime Semantics	9
3.6	Construction of XML Data	10
3.6.1	Constructing XML Instances from External Sources	10
3.6.2	Inline Construction of XML	10
3.7	Namespaces	11
3.8	Implicit Coercions	11
3.9	Output of XML Data	13
3.10	Exception Handling	13
4	Limitations	13
4.1	Unsupported XML Schema Features	13
4.2	Limitations in support of Java	14
4.3	Updates of XML Data	14
5	Coming Attractions	14
A	Example	15
A.1	XML Schema – salesschema.xsd	15

A.2 XML Data File – chart.xml	15
A.3 Program Description	17
A.4 Totals: A complete run	21
B XML Schema Built-in Types	22
C Compiler Options	23
C.1 Classpath options	23
C.2 Compliance options	23
C.3 Warning options	23
C.4 Debug options	24
C.5 Advanced options	24
C.6 XJ-specific options	24
C.7 General options	24
C.8 Script options	24
D Runtime Options	25
D.1 Standard java options	25
D.1.1 Assert options	25
D.2 Script options	25
E Apache Ant Integration	26
F Platform Notes	27
F.1 Linux	27
F.2 Cygwin	27
F.3 Windows	27
F.4 Mac OS X	27
G XJ Distribution Details	28

1. INTRODUCTION

XML has emerged as the *de facto* standard for data integration; it has become an integral part of diverse application domains such as databases, messaging systems, Web Services, etc. Despite the popularity of XML, current mechanisms for developing XML-based applications in Java are low level. For example, APIs such as W3C Document Object Module (DOM) [World Wide Web Consortium 2000] or Simple API for XML (SAX) [SAX], provide minimal support for ensuring that programs are correct with respect to the XML Schemas [Thompson et al. 2004] governing the XML data. With data-binding approaches such as Java Architecture for XML Binding (JAXB) [Vajjhala and Fialli 2004], a programmer must understand how the specification maps XML Schemas to JavaTM classes and cannot program purely in terms of the XML data. Furthermore, there is no support for ensuring the correctness of XPath expressions [Clark and DeRose 1999] evaluated against documents. In these systems, a mistyped XPath expression (where one of the element names is misspelled) would not raise an error at run time — it would silently return no results. Finally, runtime library approaches to XML, such as DOM, cannot take advantage of compiler techniques in order to optimize the performance of XML processing. The high cost of processing XML is a common complaint. XML Enhancements to JavaTM, or XJ, is a research language designed to facilitate the development of applications that process XML data. It is a *strict* extension to Java 1.4 — programs that are valid according to the Java Language Specification 1.4 [Gosling et al. 2000] are valid XJ programs; they have identical behavior in both languages. XJ adds a few constructs to Java to facilitate XML processing:

- The semantics of the **import** statement is extended to allow for the import of XML Schemas, in addition to packages and types. Element declarations in an imported XML Schema are available to a programmer as classes.
- Programmers may write XPath expressions inline to navigate and extract information from XML data. The XJ compiler type checks XPath expressions with respect to appropriate XML Schema types to detect errors.
- Programmers may construct XML data by writing literal XML inline. Again, the compiler ensures that the constructed XML is valid with respect to appropriate XML types.
- The result of compiling an XJ class is standard bytecode that can be executed on any Java Virtual Machine.

In subsequent sections, we describe how to install the XJ system, compile and execute programs (§2). We provide an overview of the constructs of the XJ programming language (§3), and discuss the limitations of the current system (§4). XJ is an evolving language and we will be adding features that address some of the current limitations (§5). We welcome any suggestions for improvement both in terms of syntax as well as language semantics.

1.1 Document Conventions

All examples in this manual refer to the **Totals.xj** sample which is provided with the XJ distribution. The XML Schema used by the sample, the **Totals.xj** XJ class, and input data are provided for reference in Appendix A.

The “**sample code**” typesetting is used to highlight sample XJ code, and the “schema construct” typesetting is used to highlight XML Schema constructs.

1.2 Online Information

To get the latest XJ information (including the latest version of this manual) on-line, access the IBM external XJ project page at <http://www.research.ibm.com/xj/>.

1.3 Reporting Bugs

Please either use the XJ forum on alphaWorks (<http://alphaworks.ibm.com/tech/xj>) or send mail to xj@watson.ibm.com.

2. GETTING STARTED

The XJ system consists of a compiler “**xjc**” (analogous to “**javac**”) and a runtime environment “**xj**” (analogous to “**java**”).

2.1 Download

The XJ distribution consists of two `.zip` archives. The `xj-bin-1.0.zip` archive contains all the files needed to compile and run XJ programs on the command line, some sample XJ programs, and the PDF version of this manual. Appendix G describes the contents of this archive in greater detail. The `xj-src-1.0.zip` archive contains modifications that were made to the Eclipse JDT compiler, on which the XJ compiler is based (it is not essential to download this archive).

2.2 Requirements

The XJ compiler and runtime system require the presence of a Java 1.4 Runtime Environment (JRE). In particular, XJ assumes that the runtime environment includes a Java API for XML processing (JAXP [Mordani and Boag 2002])-compliant XML parser and the Xalan XSLT engine, both of which are distributed with standard Java 1.4 runtime environments.

We have tested the XJ system with both the IBM and the Sun JREs, under Windows/CMD, Cygwin, and Linux — if there are problems, please file a bug report (see section 1.3).

2.3 Installation

To install the XJ distribution, extract the downloaded zip. In this document, we will refer to the `XJJD/` subdirectory that will be created by the extraction as `$XJ_HOME` (Note that there is no need to actually set an environment variable `XJ_HOME`). You may wish to add `$XJ_HOME/bin` to your `PATH` environment variable to avoid specifying the full path to the XJ scripts each time.

The XJ distribution provides two executables: “`xj`” and “`xjc`”, by analogy to “`java`” and “`javac`”. Both of these executables use the following mechanisms to discover the location of the JRE installation on a machine:

- First, the `JAVA` environment variable is used for the location of the actual “`java`” executable.
- Next, the `JAVA_HOME` environment variable is used for the location of the JRE installation directory.
- Finally, the `PATH` environment variable is used to find the “`java`” executable.

2.4 Compiling an XJ Program

XJ programs can be compiled by running “`xjc <filename>`”. The XJ compiler recognizes files with either the “`.java`” and the “`.xj`” extensions. Most of the options available in a standard `javac` compiler are available in the XJ compiler `xjc`. For a full list of available options, run “`xjc -help`”. For example, one can specify additional classpath entries by supplying one or more “`-cp <directory_or_jarfile>`” options to `xjc` (these options are cumulative), or by setting the `$CLASSPATH` environment variable. As in Java, the classpath entries are used to find classes, packages, and other resources. In addition, the classpath entries are used to locate XML Schemas referred to by a program (§3.1.1).

For example (on Linux), to compile the class `Totals` provided in the `samples` directory of the distribution:

```
[$XJ_HOME/samples]$ export JAVA_HOME=/opt/IBMJava2-141/
[$XJ_HOME/samples]$ ../bin/xjc com/ibm/xj/samples/totals/Totals.xj
[$XJ_HOME/samples]$
```

In the example, the user sets the `JAVA_HOME`¹ environment variable to refer to the appropriate directory, and then compiles the XJ class `Totals.xj`. Note that successful compilation should produce no output.

By default, `xjc` places generated class files in the same directory as the source that is compiled (as does `javac`). This setting can be changed by using the “`-d <directory>`” `xjc` option. The generated class files are as defined by Java and can be executed on any Java Virtual Machine.

All references in a program to XML Schema constructs are removed in the compilation process and are not present in the generated class file. As a result, separate compilation of XJ classes may not work properly. When compiling an XJ class that refers to another class, `B`, the XJ compiler executes the following algorithm:

- (1) Source file `B.java` exists: If there exists a class `B.class` that has been modified more recently than `B.java`, then the class `B.class` is used. Otherwise, `B.java` is compiled.

¹Note: on Linux, `JAVA_HOME` is usually set correctly automatically (by a `/etc/profile.d` script).

- (2) Source file `B.xj` exists: The file is compiled irrespective of the existence of a corresponding class file. Note that if both `B.java` and `B.xj` are found, the rule corresponding to `B.java` is executed.

While the XJ compiler attempts to discover and recompile all necessary XJ classes, it is best to recompile all XJ classes that are part of a program explicitly by passing them as arguments to `xjc`.

2.5 Running an XJ program

XJ programs are compiled to standard Java class files that can be executed on any Java Virtual Machine. To execute properly, these classes require certain JARs (corresponding to the XJ runtime system) to be in the classpath at runtime. The “`xj`” executable invokes “`java`” with the appropriate classpath settings. Run “`xj -help`” for a list of available options.

For example (on Linux), to run the `Totals` class compiled in the previous step, one invokes the `xj` executable with the appropriate arguments.

```

[$XJ_HOME/samples]$ ../bin/xj com.ibm.xj.samples.totals.Totals com/ibm/xj/samples/totals/chart.xml
Total Sales
1998 72.0
1999 79.0
2000 88.0
<?xml version="1.0" encoding="UTF-8"?>
<sales unit="GBP">430.2</sales>
Grand Total: 239.0
[$XJ_HOME/samples]$

```

3. XJ LANGUAGE OVERVIEW

The XJ programming language adds a set of classes to the standard class library defined by Java (See Figure 1). At the root of this set of classes is the `XMLObject` class, which corresponds loosely to the `Node` class in DOM [World Wide Web Consortium 2000]. The `XMLObject` class and its subclasses are treated specially by the XJ compiler:

- XPath expressions can be evaluated only on instances of these classes.
- Instances of these classes may be constructed from literal XML.
- XJ extends the Java type system to allow programmers to declare variables, methods, and fields using element declarations imported from XML Schema declarations. Element declarations in XML Schema are translated to subclasses of `XMLObject`.

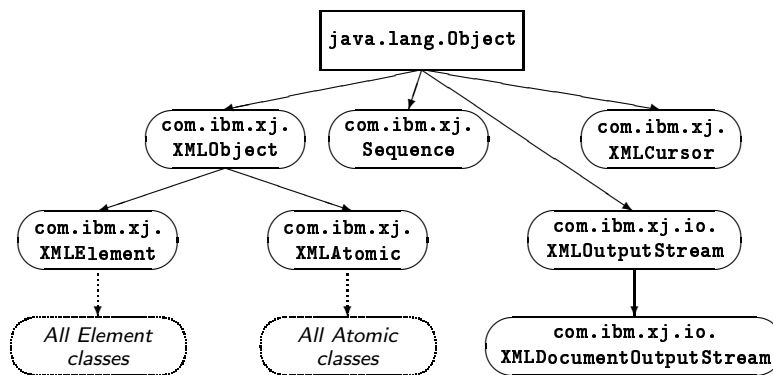


Fig. 1. The hierarchy of XJ types

In the class hierarchy, **XMLElement** is the superclass of all classes corresponding to XML element declarations, and **XMLAtomic** is the superclass of all classes corresponding to XML atomic types. The **XMLObject** and **XMLAtomic** classes are abstract and instances of them cannot be constructed directly. Instances of **XMLElement** may be constructed, as described in §3.6.

The result of an XPath expression is always an instance of a **Sequence** class, which corresponds to an ordered list of zero-or-more **XMLObject**. The **XMLCursor** class implements the `java.util.Iterator` interface, and is used to iterate over instances of the **Sequence** class. Both of these classes support a limited form of genericity as defined in Java 5.0 [Sun Microsystems] for more robust typechecking. The classes in the `com.ibm.xj.io` package support serialization of XML data to `java.io.OutputStream`.

In this section, we will describe the XJ classes, XPath expressions, and the construction of XML data in greater detail. We will also describe the interaction between XML classes and the types of Java — the coercions between XML Schema types (such as `xsd:int`) and corresponding primitive types (such as `int`).

3.1 XJ Type References

The XJ language allows programmers to refer to element declarations defined in XML Schemas as if they were classes. In addition, XJ programmers may refer to the XML Schema built-in types [Biron and Malhotra 2004] as classes as well.

3.1.1 Referring to XML Schemas. An XML Schema in XJ is treated in a manner similar to a package in Java. When resolving a type, the XJ compiler first attempts to resolve a simple name using the standard mechanisms of Java [Gosling et al. 2000, §6]. If a corresponding package or type cannot be found, the compiler appends “.xsd” to the simple name and attempts to discover an XML Schema of that name using the current classpath. For example, consider the type reference:

```
com.ibm.xj.samples.totals.salesschema
```

Assume the compiler has found a package corresponding to `com.ibm.xj.samples.totals`. The compiler will then attempt to discover an appropriate package or type named `salesschema` in the package `com.ibm.xj.samples.totals`. If this fails, the compiler will try to find the resource with the name `com/ibm/xj/samples/totals/salesschema.xsd` using the standard resource mechanism of Java (that is, using the classpath).

3.1.2 Referring to Element Declarations. Element declarations in XML Schema may be global or local in scope. Global element declarations appear at the top level of the XML Schema document. Local element declarations, on the other hand, appear entirely within a complex type definition. Names used in local element declarations are scoped to the complex type within which they are declared. For example, in the XML Schema defined in Appendix A, the declaration of element `salesdata` is global. Local declarations of a `sales` element occur in the definitions of the complex types `YearType` and `RegionType`.

Global element declarations in an XML Schema are treated in XJ as top-level classes in a package. So for example, the reference:

```
com.ibm.xj.samples.totals.salesschema.salesdata
```

would refer to the XJ class corresponding to the global element declaration of `salesdata` in the XML Schema, `salesschema`.

Local element declarations are named as if they were nested classes. Element names of local scope are disambiguated by qualifying the names with the sequence of names of containing elements, where each name in the sequence is separated by a “.”. The following type refers to the `sales` element that is defined within the `year` local element declaration in `salesdata`.

```
com.ibm.xj.samples.totals.salesschema.salesdata.year.sales
```

The following type refers to a different `sales` element declaration, the one that is defined in the scope of `RegionType` (instances of these types are not assignable to each other):

```
com.ibm.xj.samples.totals.salesschema.salesdata.year.region.sales
```

Two types corresponding to elements in an XML Schema are the same if they refer to the same element declaration in the XML Schema. Note that there may be multiple ways of specifying the same XJ type. All XJ XML classes derived from element declarations are subclasses of the **XMLElement** class.

3.1.3 Schema Built-In Types. XJ supports the built-in atomic types, such as `xsd:string` and `xsd:decimal`, defined by XML Schema. Appendix B lists all the built-in types defined by XML Schema. To refer to an XML Schema built-in atomic type in an XJ program, the type name is prefaced by “`com.ibm.xj.xsd.`” or just by “`xsd.`”. For example, `xsd.integer` refers to the XML Schema built-in type `xsd:integer`.

All the subtyping relationships between types listed in Appendix B are supported by XJ. For example, the following assignment is valid because `xsd:short` is a subtype of `xsd:decimal`:

```
xsd.short shortVar = ...;
xsd.decimal decVar = shortVar;
```

The simple types in Appendix B, `xsd:NMTOKENS`, `xsd>IDREFS`, and `xsd:ENTITIES` are represented using the **Sequence** (§3.3) class in XJ. For example, the built-in list type `xsd:NMTOKENS` is represented as the XJ type `Sequence<xsd.NMTOKEN>`.

Currently, construction of XML atomic types and references to user-defined atomic and simple types is not supported.

3.2 Import of XML types

XJ modifies the semantics of the **import** statement (the syntax remains unchanged). In addition to type-on-demand imports of packages and single-type-imports, XJ supports type-on-demand imports of XML Schemas, and single-type-imports of XML element and atomic type declarations in XML Schemas using the lookup mechanism described in §3.1.1.

The rules for disambiguation of names are as in Java. Local names hide imported names, and if the same simple name is used for two different imported types, a compile-time error occurs. The type-import-on-demand declaration of an XML Schema behaves as that of a type-import-on-demand of a package in Java. Specifically, the same disambiguation rules hold if two XML Schemas (or an XML Schema and a package) declare types whose names conflict.

3.2.1 Type-on-demand Import of an XML Schema. Consider an on-demand **import** statement of the form (as in Java) :

```
import PackageOrTypeName.*;
```

The XJ compiler first attempts to resolve *PackageOrTypeName* using the standard mechanisms used by Java. If a corresponding package or type cannot be found, the compiler appends “.xsd” to *PackageOrTypeName* and attempts to discover an XML Schema of that name using the mechanism of §3.1.1. If such an XML Schema is discovered, all global element declarations in the XML Schema are available within the compilation unit. Programmers can declare variables, methods, and fields using these types; they may be used wherever a reference type is expected. For example, given an import-on-demand statement of `salesschema`, the use of the simple name `salesdata`, which refers to the global element declaration in `salesschema`, is valid:

```
import com.ibm.xj.samples.totals.salesschema.*;
:
salesdata document;
```

3.2.2 Single-type-import of XML Schema Declarations. A single type **import** statement with respect to an XML Schema behaves as expected, that is, only the declaration referred to by the statement is visible within the compilation unit. In the example below, the first **import** statement imports only the global element `salesdata` from the XML Schema `salesschema`, and the second **import** statement imports the XJ type corresponding to the `sales` element defined in `YearType`.

```
import com.ibm.xj.samples.totals.salesschema.salesdata;
import com.ibm.xj.samples.totals.salesschema.salesdata.year.sales;
```

The following statement imports a different `sales` element declaration, the one that is defined in the scope of `RegionType` (instances of the two `sales` types are not assignable to each other):

```
import com.ibm.xj.samples.totals.saleschema.salesdata.year.region.sales;
```

3.2.3 Default Imports. The XJ compiler automatically adds an import of the form `com.ibm.xj.*` to all compilation units (in addition to the automatic import of `java.lang.*`). A programmer may therefore refer to `XMLObject`, `Sequence`, and other classes in the default XJ package without the “`com.ibm.xj`” prefix.

3.3 XJ Generics

For better static typing of XPath expressions, XJ supports a limited form of generic types based on Java 5.0 [Sun Microsystems]. The `Sequence` and `XMLCursor` types are generic container classes, and may be parametrized with respect to other XJ types. So, for example, the type of an ordered sequence that contains `item` elements would be `com.ibm.xj.Sequence<item>` (or `Sequence<item>` for short).

The `Sequence` class is used to hold lists of XML values — the result of the evaluation of an XPath expression is always a `Sequence`. A `Sequence<E>` class, where `E` is the formal type parameter, has the following methods:

- `void add(E object)`: Adds `object` of type `E` to the sequence.
- `E get(int i)`: Retrieves the `i`th element in the sequence. As in Java, the first element is numbered 0.
- `boolean isEmpty()`: Returns `true` if the sequence is empty.
- `int size()`: Returns the number of items in the sequence.
- `XMLCursor<E> iterator()`: Returns an appropriately parameterized instance of `XMLCursor`.

The `XMLCursor` implements the `java.util.Iterator` interface, and is used to iterate over `Sequence`. The subtyping rules for generics in Java 5.0 apply for these two classes. Specifically, an instance of `Sequence<XMLObject>` cannot be assigned to an instance of `Sequence<XMLObject>`, even though `XMLObject` is a subclass of `XMLObject`.

3.4 Automatic Unboxing

To simplify programming, XJ supports automatic unboxing of `Sequence` and XJ element classes. An assignment of an XML sequence class to an XML element class is allowed statically if the parametric type of the `Sequence` is compatible with the XML element class. For example, in the following:

```
salesdata s;  
Sequence <salesdata> seq = ...;  
s = seq;
```

The assignment `s = seq`; will be valid statically because the parametric type of the sequence and the type of the element class are compatible (they are both `salesdata`). The dynamic semantics of the unboxing is to retrieve the element contained in the sequence `seq` if `seq` is a singleton sequence, and to throw a `com.ibm.xj.NonSingletonSequenceException` otherwise.

Similarly, the compiler allows automatic unboxing of an element class to an atomic class, if according to the XML Schema declaration of the element, the element has atomic content compatible with the atomic class. For example, consider the statements:

```
salesdata.year.theyear y = ...;  
xsd.string s = y;
```

where `y` is an XJ variable declared to be of the XML element class `salesdata.year.theyear`. Since the content of `theyear` is declared to be `xsd:string`, the compiler allows the assignment. At runtime, `s` would be set to refer to the contents of the element referred to by `y`.

3.5 XPath Expressions

The ability to specify XPath expressions inline in XJ enables programmers to extract information from XML data in a concise and declarative manner. Most of the processing of XML data in XJ is done with XPath

expressions; in XJ, the expression syntax is extended to include XPath expressions. The syntax for specifying XPath expressions in XJ is all expressions that satisfy the production **XPath** defined as follows:

```
XPath ::= QualifiedIdentifier '[' XP ']'
       ::= Primary '[' XP ']'
XP     ::= Expr
```

QualifiedIdentifier [Gosling et al. 2000, §18.1, page 449] and **Primary** [Gosling et al. 2000, §18.1, page 451] refer to the corresponding productions defined by the Java Language Specification. **XP** corresponds to an **Expr**, where this non-terminal is as defined by XPath 1.0 [Clark and DeRose 1999, §3.1]. We will refer to the **QualifiedIdentifier** or **Primary** with respect to which the XPath expression is evaluated as the *context specifier*. The XJ parser lexically analyzes XPath expressions as specified by the XPath specification; keywords and other parsing constructs of Java do not apply while parsing an XPath expression.

3.5.1 Referring to In-Scope Variables. The XPath expression can refer to XJ variables that are visible in the current scope. The following sample code demonstrates the use of an XPath expression that refers to an XJ variable **min**. At runtime, the XPath expression would return all **year** children of the context node (the value referred to by **sd**) such that the sum of the contents of the **sales** descendants of the **year** element is greater than the value of **min**.

```
int min = 70;
Sequence<year> ys = sd[|year[sum(./sales) > $min]|];
```

3.5.2 Static Semantics. An XPath expression is valid in XJ if the type of the context specifier of an XPath expression resolves to a subclass of **XMLObject** or to a **Sequence**. It is a static error if the evaluation of the XPath expression returns an empty result for all XML values that are valid with respect to the type of the context node specifier. For example, the XPath expression in the following code would raise a static error since **year** elements do not have children labeled **sale** and the compiler can determine that the XPath expression will always return an empty result set at runtime.

```
year y = ...;
Sequence s = y[|sale|];
```

All variables used in the XPath expression must be visible in the current scope. Only variables whose names correspond to the identifier syntax of Java are considered valid; specifically, variables whose names contain namespace references will raise a compile-time error. The result type of an XPath expression is always an instance of the **com.ibm.xj.Sequence<...>** type. If the compiler can determine that the XPath expression always evaluates to instances of a particular XML class, it returns a **Sequence** of that type. Otherwise, it returns a **Sequence** of **XMLObject**. So, for example, in the XPath expression below, the static type of the XPath expression is determined to be **Sequence<year>**, since the XPath expression will always return zero-or-more **year** elements.

```
int min = 70;
Sequence<year> ys = sd[|year[sum(./sales) > $min]|];
```

A unique type cannot be determined for the result of the following XPath expression; the static type of the XPath expression is **Sequence<XMLObject>**.

```
sd[|/*|];
```

3.5.3 Runtime Semantics. At runtime, the XPath expression is evaluated with respect to the XML value referred to by the context specifier as defined by XPath 1.0. If the context specifier evaluates to a **Sequence** at runtime, each member of the **Sequence** is used as a context node in the evaluation of the XPath expression, and the union of the results of all evaluations are taken to form the result. The result is an instance of **Sequence** — an ordered list of zero-or-more XML values with no duplicates. According to the XPath 1.0 specification, an XPath expression can return a *node set*, *number*, *boolean value*, or *string*. When the result of the XPath expression is a *node set*, the XJ result of the XPath expression is a **Sequence** consisting of the nodes in the set in arbitrary order. When result of the XPath expression is not a node set, the XJ result of

the XPath expression is a **Sequence** of the appropriate type (for example, **Sequence**<`xsd:boolean`>, when the result is a *boolean value*). In XPath evaluation, the current values of variable references are used.

3.6 Construction of XML Data

XJ introduces two mechanisms for constructing instances of XML classes that correspond to XML Schema element type declarations. XML data can be constructed either from an external source or by embedding literal XML in an XJ program.

3.6.1 *Constructing XML Instances from External Sources.* **XMLElement** and all XML classes corresponding to global element declarations have three constructors:

```
—salesdata(java.io.InputStream)
—salesdata(java.io.File)
—salesdata(java.net.URL)
```

where **salesdata** is an XJ class derived from an XML Schema element declaration. These constructors load the XML data from the stream, file, or URL as appropriate and construct appropriate instances of the XJ classes corresponding to the elements in the data. For example, following expressions load an XML value and construct instances of **salesdata**:

```
new salesdata(new java.io.FileInputStream("com/ibm/xj/samples/totals/chart.xml"));
new salesdata(new java.io.File("com/ibm/xj/samples/totals/chart.xml"));
new salesdata(new java.net.URL("file:com/ibm/xj/samples/totals/chart.xml"));
```

3.6.2 *Inline Construction of XML.* XJ supports the construction of XML data using literal XML — the syntax of this construction is similar to that of direct element construction in XQuery [Katz et al. 2003]. For example, the following statement creates an instance of the **theyear** element class:

```
theyear y = new theyear(<theyear>1998</theyear>);
```

The XML literal within the parentheses can be any well-formed block of XML:

```
region r = new region(<region>
    <name>NorthEast</name>
    <sales unit='GBP'>75</sales>
</region>);
```

The content of the XML literal is validated according to the XML Schema declaration for the declared element type, in this case, **region**. One can construct dynamic XML based on expressions evaluated at runtime. For example, given the previous two statements, one can construct an instance of **salesdata**:

```
float conversion = 1.9;
salesdata s =
    new salesdata(
        <salesdata>
            <year>
                {y}
                <sales unit='Dollars'>{grossSales * conversion}</sales>
                {r}
            </year>
        </salesdata>);
```

The braces, '{' and '}' delimit XJ expressions that are evaluated at runtime to provide the values that are to be inserted during construction. The pattern '{{' is interpreted as a single '{' in case the programmer wishes to insert a literal '{'. Similarly '}}' is interpreted as '}'². In the example, **grossSales** is some XJ

²Note that braces can also be inserted using the '{' and '}' constructs.

variable that is visible in the current scope. The result of the evaluation of the construction, assuming that the runtime value of `grossSales` is `100` is:

```
<salesdata>
  <year>
    <theyear>1998</theyear>
    <sales unit='Dollars'>190.0</sales>
  <region>
    <name>NorthEast</name>
    <sales unit='GBP'>75</sales>
  </region>
</year>
</salesdata>
```

To construct untyped XML, that is, XML that is not to be validated with respect to any XML Schema, one can use the XML literal constructor defined for `XMLElement`:

```
XMLElement a = new XMLElement(<theyear>1998</theyear>);
```

where the argument to the constructor is any well-formed block of XML data. The type of the XML value constructed is `XMLElement`. The variable `a` cannot be assigned to the variable `y` declared above even though they are constructed from the same literal XML. The variable `a` is an instance of `XMLElement`, and the variable `y` is an instance of `theyear`, and an `XMLElement` may not be assigned to a `theyear`.

3.7 Namespaces

XJ supports the declaration of XML namespaces — an XJ reference to an XML Schema can be used as the namespace prefix for the target namespace associated with the schema. So, if `sampleschema` is a reference to an XML Schema that is associated with the target namespace, “`http://sample.com/sampleschema`”, “`sampleschema`” can be used as the namespace prefix corresponding to that target namespace in XPath expressions and construction.

For example, the following code constructs an element `book` in the target namespace associated with `sampleschema`. Note the use of `xmlns` attributes to declare new namespace prefixes (as in XML).

```
import sampleschema.book;
...
new book(<sampleschema:book>
  <au:author xmlns:au="http://sample.com/author">
    John Steinbeck
  </au:author>
</sampleschema:book>
);
```

The following code would be invalid since the `<book>` element constructed is in the default namespace and `sampleschema.book` is in the namespace associated with `sampleschema`, “`http://sample.com/sampleschema`”

```
import sampleschema.book;
...
new book(<book>
  <au:author xmlns:au="http://sample.com/author">
    John Steinbeck
  </au:author>
</book>
);
```

3.8 Implicit Coercions

XJ supports a number of implicit coercions between Java types and XML types to simplify programming. The rules for automatic coercion from an XML atomic built-in type to a Java type are summarized in the

table on page 12. The XJ classes for the built-in types `xsd.QName`, `xsd.NOTATION`, `xsd.base64Binary`, `xsd.hexBinary` and `xsd.anyURI` do not have any implicit coercions defined. A value of one of these built-in atomic types may be assigned to an XML variable of the same type or used in the construction of an element type instance.

Consider primitive XML Schema types. An implicit coercion of the primitive XML Schema type `xsd.float` to its corresponding primitive Java type `float` is allowed. Similarly, an implicit coercion of `xsd.double` to the primitive Java type `double` is allowed, and of `xsd.boolean` to the primitive Java type `boolean`. An implicit coercion of the primitive XML Schema type `xsd.decimal` to the primitive Java type `double` is allowed.

The primitive XML Schema type `xsd.string` is a finite length sequence of characters. The atomic class corresponding to `xsd.string` can be implicitly coerced into the Java class `java.lang.String`, which is a fixed length sequence of the primitive type `char`, which is a 16-bit unsigned integer representing a Unicode character.³

The XML Schema built-in numeric types `xsd:byte`, `xsd:short`, `xsd:int`, and `xsd:long` derive from the built-in type `xsd:integer`. Each of these derived XML Schema types can also be implicitly coerced to the corresponding Java primitive class of the same name. `xsd:integer` in turn derives from the primitive type `xsd:decimal` and can also be implicitly coerced to the Java class `long`.

XML Schema type	Java type	XML Schema type	Java type
<code>xsd:anySimpleType</code>	<code>Sequence<XMLObject></code>	<code>xsd:byte</code>	<code>byte</code>
<code>xsd:float</code>	<code>float</code>	<code>xsd:nonPositiveInteger</code>	<code>long</code>
<code>xsd:double</code>	<code>double</code>	<code>xsd:negativeInteger</code>	<code>long</code>
<code>xsd:boolean</code>	<code>boolean</code>	<code>xsd:nonNegativeInteger</code>	<code>long</code>
<code>xsd:string</code>	<code>java.lang.String</code>	<code>xsd:positiveInteger</code>	<code>long</code>
<code>xsd:normalizedString</code>	<code>java.lang.String</code>	<code>xsd:unsignedLong</code>	<code>long</code>
<code>xsd:token</code>	<code>java.lang.String</code>	<code>xsd:unsignedInt</code>	<code>long</code>
<code>xsd:language</code>	<code>java.lang.String</code>	<code>xsd:unsignedShort</code>	<code>long</code>
<code>xsd:Name</code>	<code>java.lang.String</code>	<code>xsd:unsignedByte</code>	<code>long</code>
<code>xsd:NMTOKEN</code>	<code>java.lang.String</code>	<code>xsd:date</code>	<code>java.util.Date</code>
<code>xsd:NCName</code>	<code>java.lang.String</code>	<code>xsd:dateTime</code>	<code>java.util.Date</code>
<code>xsd:ID</code>	<code>java.lang.String</code>	<code>xsd:gMonthDay</code>	<code>java.util.Date</code>
<code>xsd:IDREF</code>	<code>java.lang.String</code>	<code>xsd:gDay</code>	<code>java.util.Date</code>
<code>xsd:ENTITY</code>	<code>java.lang.String</code>	<code>xsd:gMonth</code>	<code>java.util.Date</code>
<code>xsd:decimal</code>	<code>double</code>	<code>xsd:gYearMonth</code>	<code>java.util.Date</code>
<code>xsd:integer</code>	<code>long</code>	<code>xsd:gYear</code>	<code>java.util.Date</code>
<code>xsd:long</code>	<code>long</code>	<code>xsd:time</code>	<code>java.util.Date</code>
<code>xsd:int</code>	<code>int</code>	<code>xsd:duration</code>	<code>long</code>
<code>xsd:short</code>	<code>short</code>		

Table I. Implicit coercions from XML built-in atomic types to Java types

Any non-primitive XML Schema atomic type `T` must derive from some XML Schema primitive type `P`. Values of the type `T` can be implicitly coerced to the same Java type as the type `P`.

In §3.4, we described how an XML element or sequence class can be unboxed to an XML atomic class. Given the rules for implicitly coercing an XML atomic type to a Java type, one can compose coercions and unboxing to implicitly coerce an XML element or sequence class to a Java type. For example, consider the XJ statement:

```
String s = sd[|theyear|];
```

where `sd` is an XJ variable declared to be of an XML element class `salesdata`. The sequence that is the result type of the XPath expression `sd[|theyear|]`, `Sequence<theyear>`, can be unboxed to the XML element class `theyear`, which in turn, can be unboxed to the XML atomic class `xsd.string`. Finally, the assignment brings about another implicit coercion, from `xsd.string` to `java.lang.String`.

As another example, consider the XJ statement:

```
double d = sd[|year/sales|];
```

³The rules for string encoding conversion are the default Java rules, which may not correspond to the XML Schema rules.

The XPath expression `/year/sales` yields a singleton sequence whose member is of element class `sales`, which has simple content based on `xsd:double`. Explicit cast operations from XML Schema to Java work as expected. So, `(String)sd[|theyear|]` results in the same coercions as `String s = sd[|theyear|]`.

3.9 Output of XML Data

XJ provides helper classes for outputting an instance of an XML class to an external `java.io.OutputStream`. These classes, `com.ibm.xj.io.XMLOutputStream` and `com.ibm.xj.io.XMLDocumentOutputStream` both support the interface offered by `java.io.PrintStream`, and behave similarly to `PrintStream` when invoked on objects that are XJ XML classes. For XJ XML classes, these two classes format the XML appropriately. The standard way of using `XMLOutputStream` is as follows:

```
XMLOutputStream out = new XMLOutputStream(System.out); // Any stream can be passed in
out.println(new XMLElement(<a> ... </a>));
```

`XMLDocumentOutputStream` behaves similarly to `XMLOutputStream` except that it prepends the XML header (for example, `<?xml version="1.0" encoding="UTF-8">`) before outputting an XML class. Only the `"UTF-8"` encoding is supported at the moment.

3.10 Exception Handling

All run-time exceptions that are XJ-specific are instances of `com.ibm.xj.XJException` or one of its subclasses. The `XJException` class is used to uniformly wrap any exceptions that may be thrown by the XJ runtime. `XJExceptions` are unchecked exceptions, so they do not need to be declared in the `throws` clause of a method. Thus, they are similar to Java's `RuntimeException`.

There is currently one subclass of `XJException`: `com.ibm.xj.NonSingletonSequenceException`, thrown when a `Sequence` with more than one element is coerced to an element type.

Here are the current scenarios that will result in throwing an instance of `XJException`:

- (1) When the XJ runtime fails to initialize.
- (2) When the XML Schema file to be imported could not be found at run time.
- (3) When a change to the imported XML Schema is detected at run time.
- (4) When the imported XML Schema file could not be opened at run time.
- (5) When constructing from a stream and the document could not be parsed.
- (6) When an XPath expression could not be evaluated.
- (7) When attempting to coerce an empty sequence or a null element to an atomic type.
- (8) When unable to convert the text content of an XML element to a given atomic type.

If an exception that is not a (subclass of) `XJException` is thrown by the XJ runtime, this is a bug and should be reported (§1.3).

4. LIMITATIONS

XJ is an evolving language and has certain limitations that we are addressing. In this section, we describe the features of XML and XML Schema that are not supported currently by XJ. XJ programs have only been tested with the standard XML encoding.

4.1 Unsupported XML Schema Features

XJ does not currently support certain XML Schema features, such as the subtyping mechanisms of type substitution and substitution groups for elements. Subtyping does work for the XML Schema built-in types. We also do not support XML Schema constructs such as `import`, `include`, and `redefine`. Identity constraints expressed using `key` and `keyref` and redefinition of declarations are not supported. Furthermore, XML class names with `".`", `"-"`, `":"`, or corresponding to a Java keyword will not be parsed appropriately. Note that the above does not apply to XML Schema primitive types expressed using the `"xsd."` notation.

With respect to validation, we do not currently validate XML data constructed from external XML sources. Construction from literal XML is validated with respect to XML Schema, but is not exhaustive in the checks

performed. For example, the validation does not currently check facet constraints or cardinality bounds, such as, `minOccurs`. Also, wildcards such as `anyattribute` in XML Schema are not yet supported.

Coercions from primitive Schema types `xsd:date`, `xsd:time`, and `xsd:dateTime` (as well as the types `xsd:gYear`, `xsd:gYearMonth`, `xsd:gMonth`, `xsd:gMonthDay`, and `xsd:gDay`, which are essentially restrictions on `xsd:date`) to `java.util.Date` are not yet implemented. Similarly, the coercion of the primitive Schema type `xsd:duration` to Java's primitive type `long` has not yet been implemented.

With respect to output of XML data, we currently only support output using the "UTF-8" encoding.

4.2 Limitations in support of Java

Currently introspection and reflection of XML classes is not supported. Similarly, downcasts to XML classes (for example, `(theyear) o`, where `theyear` is an XML class and `o` is of type `Object`) are not supported either. The expressions involving XML classes, that is, XPath evaluation and construction, are not guaranteed to be thread safe.

4.3 Updates of XML Data

Updates of XML data are not supported currently. It will be supported shortly in a new release.

5. COMING ATTRACTIONS

We will support in-place updates of XML data through the use of XPath expressions on the left-hand side of assignment statements. We will providing WSDL support for easy development of Web Services. Finally, we are developing an Eclipse plugin to simplify development of XJ applications.

REFERENCES

- APACHE SOFTWARE FOUNDATION. 2005. *Apache Ant*. <http://ant.apache.org/>.
- BIRON, P. V. AND MALHOTRA, A., Eds. 2004. *XML Schema Parts 2: Datatypes Second Edition*. World Wide Web Consortium.
- CLARK, J. AND DEROSE, S., Eds. 1999. *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium.
- GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc. <http://java.sun.com/j2se/1.4.2/>.
- KATZ, H. ET AL. 2003. *XQuery from the Experts. A Guide to the W3C XML Query Language*. Addison Wesley.
- MORDANI, R. AND BOAG, S. 2002. *Java API for XML Processing Version 1.2*. Sun Microsystems.
- SAX. *Simple API for XML*. <http://www.saxproject.org/>.
- Sun Microsystems. *JDK 5.0 Documentation*. Sun Microsystems. <http://java.sun.com/j2se/1.5.0/docs/index.html>.
- THOMPSON, H. S., BEECH, D., MALONEY, M., AND MENDELSON, N., Eds. 2004. *XML Schema Parts 1: Structures Second Edition*. World Wide Web Consortium.
- VAJJHALA, S. AND FIALLI, J., Eds. 2004. *The Java Architecture for XML Binding (JAXB) 2.0*. Sun Microsystems.
- World Wide Web Consortium 2000. *Document Object Model Level 2 Core*. World Wide Web Consortium.

A. EXAMPLE

We now illustrate the use of the XJ extensions of Java using a sample XJ program. We first list the XML Schema imported by the program and the data processed by the program. We then describe in detail each construct of the sample program. The sample program **Totals.xj**, the imported XML Schema, and the XML data file can be found in the distribution in the sub-directory `samples/com/ibm/xj/samples/totals`.

A.1 XML Schema – saleschema.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="salesdata">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="year" type="YearType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="YearType">
    <xsd:sequence>
      <xsd:element name="theyear" type="xsd:string"/>
      <xsd:element name="sales" type="SalesType"/>
      <xsd:element name="region" type="RegionType" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="SalesType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:double">
        <xsd:attribute name="unit" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="RegionType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="sales" type="SalesType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

A.2 XML Data File – chart.xml

```
<?xml version="1.0"?>
<salesdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="salesdata.xsd">
  <year>
    <theyear>1997</theyear>
    <region>
      <name>west</name>
      <sales unit="millions">32</sales>
    </region>
    <region>
      <name>central</name>
```

continued...

```
    <sales unit="millions">11</sales>
  </region>
  <region>
    <name>east</name>
    <sales unit="millions">19</sales>
  </region>
</year>
<year>
  <theyear>1998</theyear>
  <region>
    <name>west</name>
    <sales unit="millions">35</sales>
  </region>
  <region>
    <name>central</name>
    <sales unit="millions">12</sales>
  </region>
  <region>
    <name>east</name>
    <sales unit="millions">25</sales>
  </region>
</year>
<year>
  <theyear>1999</theyear>
  <region>
    <name>west</name>
    <sales unit="millions">36</sales>
  </region>
  <region>
    <name>central</name>
    <sales unit="millions">12</sales>
  </region>
  <region>
    <name>east</name>
    <sales unit="millions">31</sales>
  </region>
</year>
<year>
  <theyear>2000</theyear>
  <region>
    <name>west</name>
    <sales unit="millions">37</sales>
  </region>
  <region>
    <name>central</name>
    <sales unit="millions">11</sales>
  </region>
  <region>
    <name>east</name>
    <sales unit="millions">40</sales>
  </region>
</year>
</salesdata>
```

A.3 Program Description

Totals.xj performs a simple data processing job. It works on a fairly familiar schema, describing sales per each year of interest, and within each year, per each region of interest. The document element is salesdata. It contains a sequence of year elements. A year element contains the year (element theyear). In addition, it contains a sequence of region elements. A region element, in turn, contains the name of the region (element name) and the sales in this region (element sales).

```
package com.ibm.xj.samples.totals;
import java.io.FileInputStream;
import com.ibm.xj.io.XMLDocumentOutputStream;
import com.ibm.xj.samples.driver.Benchmark;
import com.ibm.xj.samples.totals.sale-schema.*;
public class Totals implements Benchmark {
    private salesdata document;

    public static void main(String[] argv) {
        if (argv.length != 1) {
            System.err.println("Usage: Totals <filename>");
            System.exit(-1);
        }
        Totals t = new Totals();
        t.doParse(argv[0]);
        t.doExecute();
    }

    public void doParse(String filename) {
        try {
            document = new salesdata(new FileInputStream(filename));
        }
        catch (java.io.IOException e) {
            throw new Error("Cannot parse input file");
        }
    }

    public void doExecute() {
        computeSales(document);
    }

    private static void computeSales(salesdata sd) {
        int min = 70;
        System.out.println("Total Sales");
        double grandTotal = 0;
        Sequence<year> ys = sd[|year[sum(../sales) > $min|]];
        if (ys.isEmpty())
            System.out.println("No elements matched");

        XMLCursor<year> y = ys.iterator();
        while (y.hasNext()) {
            year ytmp = y.next();
            String str = ytmp[|theyear|];
            System.out.print(str + "\t");
            double total = 0;
            XMLCursor<sales> s = ytmp[|../sales|].iterator();
```

continued...

```

        while (s.hasNext()) {
            sales stmp = s.next();
            total = total + stmp;
        }
        grandTotal += total;
        System.out.println(total);
    }
    double conversionFactor = 1.8;
    sales s2 = new sales(<sales unit="GBP">{conversionFactor * grandTotal}</sales>);

    XMLDocumentOutputStream out = new XMLDocumentOutputStream(System.out);
    out.println(s2);

    System.out.println("\n Grand Total: " + grandTotal);
}
}

```

We now examine this program, statement by statement, and explain the various XJ features that are employed. The preamble:

```
package com.ibm.xj.samples.totals;
```

The above specifies the package in which the Java class resides.

```
import java.io.FileInputStream;
import com.ibm.xj.io.XMLDocumentOutputStream;
import com.ibm.xj.samples.driver.Benchmark;
```

As an XJ program is a Java program, one can import any Java classes and interfaces.

```
import com.ibm.xj.samples.totals.sale-schema.*;
```

This statement notifies the XJ compiler that the XML schema entities (elements and their types) in the file “`sale-schema.xsd`” in package `com.ibm.xj.samples.totals` should be imported into the program.

```
public class Totals implements Benchmark {
    private salesdata document;
```

The file defines a Java class — **Totals** that implements a Java interface **Benchmark**⁴. Within this class, the first declaration is of a private variable named **document**. Its type is **salesdata**, which is an XML type. The XJ compiler will find the definition of salesdata in the imported schema file.

```

    public static void main(String[] argv) {
        if (argv.length != 1) {
            System.err.println("Usage: Totals <filename>");
            System.exit(-1);
        }
        Totals t = new Totals();
        t.doParse(argv[0]);
        t.doExecute();
    }
}

```

This main method creates an object of class **Totals** and asks it to perform two tasks:

- parse the filename passed on the command line by calling instance method **doParse**, and
- perform its data processing tasks by calling instance method **doExecute**.

⁴A utility interface for micro-benchmarking

Incidentally, the two methods above are defined in the **Benchmark** interface.

```
public void doParse(String filename) {
    try {
        document = new salesdata(new FileInputStream(filename));
    }
    catch (java.io.IOException e) {
        throw new Error("Cannot parse input file");
    }
}
```

Consider the statement `document = new salesdata(new FileInputStream(filename));`. It performs loading, parsing and validation⁵ of the data in file `filename` and associates an XML value with the XML variable `document`.

```
public void doExecute() {
    computeSales(document);
}
```

Method `doExecute` performs the data processing task by invoking the `computeSales` static method on the `document` field.

```
private static void computeSales(salesdata sd) {
    int min = 70;
    System.out.println("Total Sales");
    double grandTotal = 0;
```

Method `computeSales` has a single argument, an XML parameter `sd` of type `salesdata`. It first prints the output heading and initializes a double value `grandTotal` to 0. Then, it performs:

```
Sequence<year> ys = sd[|year[sum(../sales) > $min]|];
```

This statement defines an XJ sequence, `ys`, parameterized⁶ by the XML type `year`. This sequence will contain, at run-time, references to XML values. It is obtained by applying an XPath expression to the XML value referenced by variable `sd` (the argument to `computeSales`). The XPath expression is `sd[|year[sum(../sales) > $min]|]`, which finds all `years` for which the sum of the values contained in the `sales` elements is greater than the value of `min`. Note that this XPath expression refers to the XJ variable `min`. At runtime, the value of `min` would be substituted before the evaluation of the XPath expression. Formally, this expression results in a sequence of XML items (all are nodes in this case). XJ's implementation is to populate the sequence with references to XML elements or instances of atomic classes as appropriate.

```
if (ys.isEmpty())
    System.out.println("No elements matched");
```

This statement checks if the above sequence is empty, and prints a message if so. The `isEmpty()` method is defined on all **Sequences**.

```
XMLCursor<year> y = ys.iterator();
```

Here, an **XMLCursor** is created for the `ys` sequence. An **XMLCursor** is very similar to the Java **Iterator**, but is generic.

```
while (y.hasNext()) {
    year ytmp = y.next();
```

⁵Validation of the document is currently disabled by default.

⁶Java 1.5 style generics are only supported for XML types at the moment.

The **while** loop iterates over the elements of **y**. These elements are references to year elements. Variable **ytmp** references the current year element.

```
String str = ytmp[|theyear|];
System.out.print(str + "\t");
double total = 0;
```

String **str** is assigned the character content of theyear sub-element of element year. The variable **total**, the sum for this year, is initialized to zero. At this point the sales of the current year element are to be added.

```
XMLCursor<sales> s = ytmp[|../sales|].iterator();
```

The **XMLCursor s** will iterate over the sales elements within the current year element.

```
while (s.hasNext()) {
    sales stmp = s.next();
```

The loop iterates through the sales elements in **s**. Variable **stmp** is the current sales element.

```
total = total + stmp;
```

The **total** for this year element is updated. Note that there's an implicit coercion from the XML element sales that has contents of type xsd:double to **double**.

```
}
grandTotal += total;
System.out.println(total);
```

The **grandTotal** (over all years) is incremented; the **total** for this year is printed.

The following demonstrates the XML value construction facilities of XJ:

```
}
double conversionFactor = 1.8;
sales s2 = new sales(<sales unit="GBP">{conversionFactor * grandTotal}</sales>);
```

It creates a new XML value using the literal XML constructor and stores in the variable **s2**. The braces “{ }” enclose an XJ expression which is evaluated at runtime to generate a value that is embedded in the constructed XML.

Variable **s2** references the newly constructed XML value. The statement illustrates one way in which data can be assigned to this element. The XML value of **s2** at runtime is:

```
<?xml version="1.0" encoding="UTF-8"?>
<sales unit="GBP">430.2</sales>
```

Next,

```
XMLDocumentOutputStream out = new XMLDocumentOutputStream(System.out);
```

A new **com.ibm.xj.io.XMLDocumentOutputStream** object is created, which adds XML output capability to the **java.io.PrintStream** class. Then,

```
out.println(s2);
```

s2 is serialized into plain XML and is printed to standard output via the **XMLDocumentOutputStream** class.

```
System.out.println("\n Grand Total: " + grandTotal);
}
}
```

Finally, “**Grand Total: 239.0**” is printed.

A.4 Totals: A complete run

The compilation and execution are as follows:

```
[XJ_HOME/samples]$ export JAVA_HOME="c:/Program Files/IBM/Java14"
[XJ_HOME/samples]$ ../bin/xjc com/ibm/xj/samples/totals/Totals.xj
[XJ_HOME/samples]$ ../bin/xj com.ibm.xj.samples.totals.Totals com/ibm/xj/samples/totals/chart.xml
Total Sales
1998 72.0
1999 79.0
2000 88.0
<?xml version="1.0" encoding="UTF-8"?>
<sales unit="GBP">430.2</sales>
Grand Total: 239.0
[XJ_HOME/samples]$
```

B. XML SCHEMA BUILT-IN TYPES

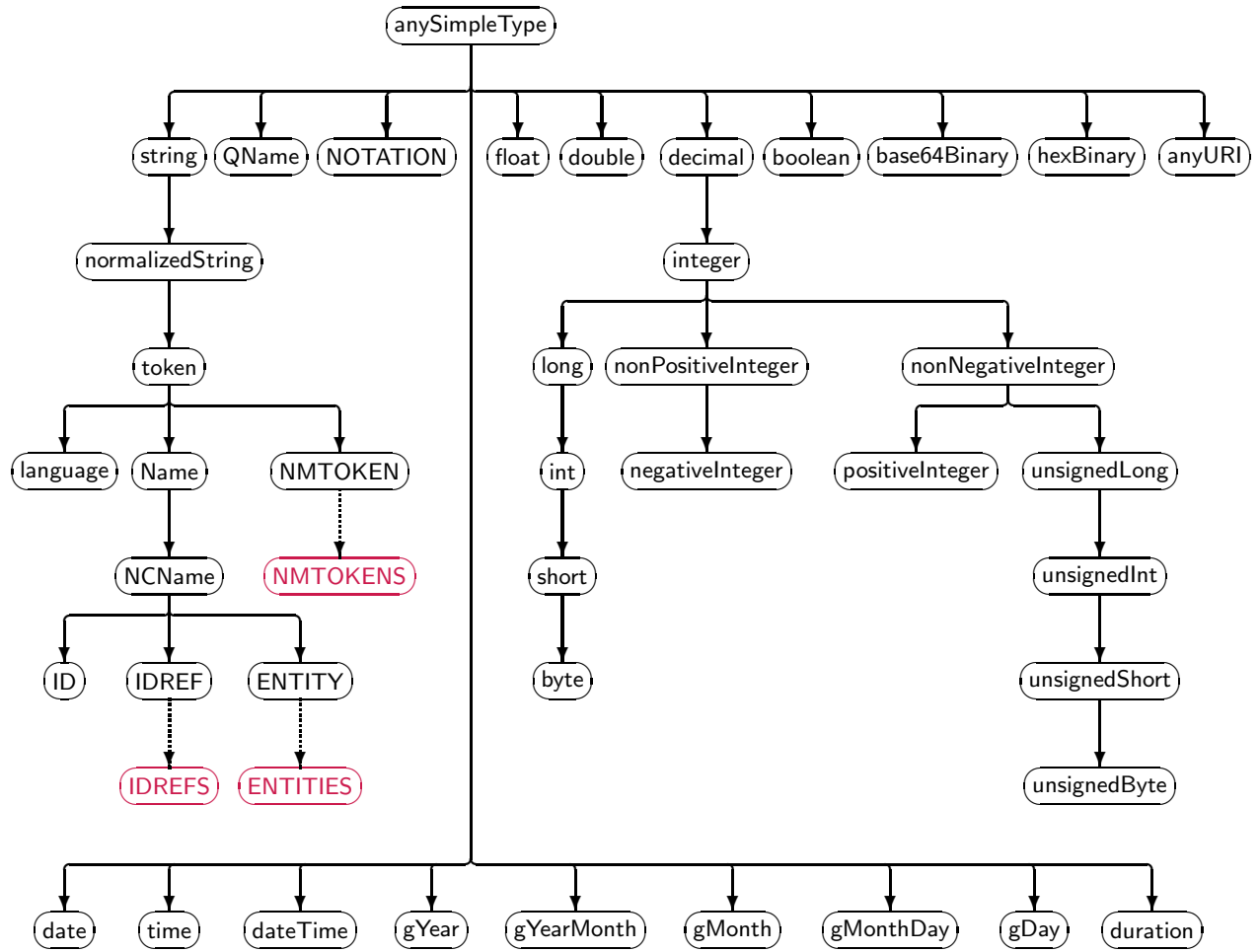


Fig. 2. The hierarchy of XML Schema built-in types

C. COMPILER OPTIONS

The `xjc` compiler currently supports the following command-line options:

C.1 Classpath options

- cp** or **-classpath** *<directories and zip/jar files separated by ;>* — specify location for application classes and sources
- bootclasspath** *<directories and zip/jar files separated by ;>* — specify location for system classes
- d** *<dir>* — destination directory (if omitted, no directory is created)
- d none** — generate no `.class` files
- encoding** *<enc>* — specify custom encoding for all sources. Each file/directory can override it when suffixed with “[*<enc>*]” (e.g. `X.java[utf8]`)

C.2 Compliance options

- 1.3** — use 1.3 compliance level (implicit `-source 1.3 -target 1.1`)
- 1.4** (default) — use 1.4 compliance level (implicit `-source 1.3 -target 1.2`)
- source** *<version>* — set source level (1.3 or 1.4)
- target** *<version>* — set classfile target (1.1 to 1.4)

C.3 Warning options

- deprecation** (default) — deprecation outside deprecated code
- nowarn** — disable all warnings
- warn:none** — disable all warnings
- warn:<warnings separated by ,>** — enable exactly the listed warnings
- warn:+<warnings separated by ,>** — enable additional warnings
- warn:-<warnings separated by ,>** — disable specific warnings

<code>allDeprecation</code> —	deprecation including inside deprecated code
<code>allJavadoc</code> —	invalid or missing javadoc
<code>assertIdentifier</code> (default) —	“ <code>assert</code> ” used as identifier
<code>charConcat</code> (default) —	<code>char[]</code> in <code>String</code> concat
<code>conditionAssign</code> —	possible accidental boolean assignment
<code>constructorName</code> (default) —	method with constructor name
<code>deprecation</code> (default) —	deprecation outside deprecated code
<code>emptyBlock</code> —	undocumented empty block
<code>fieldHiding</code> —	field hiding another variable
<code>finally</code> (default) —	finally block not completing normally
<code>indirectStatic</code> —	indirect reference to static member
<code>intfNonInherited</code> (default) —	interface non-inherited method compatibility
<code>javadoc</code> —	invalid javadoc
<code>localHiding</code> —	local variable hiding another variable
<code>maskedCatchBlock</code> (default) —	hidden catch block
<code>nls</code> —	string literal lacking non-nls tag <code>//\$NON-NLS-<n>\$</code>
<code>noEffectAssign</code> (default) —	assignment without effect
<code>pkgDefaultMethod</code> (default) —	attempt to override package-default method
<code>semicolon</code> —	unnecessary semicolon, empty statement
<code>unqualifiedField</code> —	unqualified reference to field
<code>unusedImport</code> (default) —	unused import declaration
<code>unusedLocal</code> —	unread local variable
<code>unusedPrivate</code> —	unused private member declaration
<code>unusedThrown</code> —	unused declared thrown exception
<code>unnecessaryElse</code> —	unnecessary else clause
<code>uselessTypeCheck</code> —	unnecessary cast/instanceof operation

<code>specialParamHiding</code> —	constructor or setter parameter hiding another field
<code>staticReceiver</code> (default) —	non-static reference to static member
<code>syntheticAccess</code> —	synthetic access for innerclass
<code>tasks(<tags separated by >)</code> —	tasks identified by tags inside comments

C.4 Debug options

`-g[:lines,vars,source]` — custom debug info
`-g:lines,source` (default) — both lines table and source debug info
`-g` — all debug info
`-g:none` — no debug info
`-preserveAllLocals` — preserve unused local vars for debug purpose

C.5 Advanced options

`@<file>` — read command line arguments from file
`-maxProblems <n>` — max number of problems per compilation unit (100 by default)
`-log <file>` — log to a file
`-proceedOnError` — do not stop at first error, dumping class files with problem methods
`-verbose` — enable verbose output
`-referenceInfo` — compute reference info
`-progress` — show progress (only in `-log` mode)
`-time` — display speed information
`-noExit` — do not call `System.exit(n)` at end of compilation (`n==0` if no error)
`-repeat <n>` — repeat compilation process `<n>` times for perf analysis
`-inlineJSR` — inline JSR bytecode
`-enableJavadoc` — consider references in javadoc

C.6 XJ-specific options

`-validate` (default) — enable validation of inline XML construction
`-novalidate` — disable validation of inline XML construction
`-nativeXPath` (default) — compile XPath to direct accesses of XML for efficiency
`-nonativeXPath` — do not compile XPath to direct accesses of XML, use Xalan to evaluate XPath
`-E` or `-shortErrors` — enable one-line error messages

C.7 General options

`-?` or `-help` — print the help message
`-version` — print compiler version
`-showversion` — print compiler version and continue

C.8 Script options

`-t` or `--trace` — Print the command being executed
`-cp` or `-classpath <PATH>` — Append `<PATH>` to the java classpath
`-P` or `-profile` or `--profile` — Run with profiling
`-J<arg>` — Pass `<arg>` to java, e.g., use `-J-verbose` to make java execution verbose
`-V` or `--version` — Same as `-version`
`-v` or `--verbose` — Same as `-verbose`
`-h` or `--help` — Same as `-help`

D. RUNTIME OPTIONS

The `xj` runtime invoker currently supports the following command-line options:

D.1 Standard java options

- cp** *or* **-classpath** *<directories and zip/jar files separated by ;>* — set search path for application classes and resources
- D<name>=<value>** — set a system property
- verbose[:class|gc|jni]** — enable verbose output
- version** — print product version
- showversion** — print product version and continue
- ?** *or* **-help** — print the help message
- X<option>** — non-standard option (use **-X** for help on those)

D.1.1 Assert options

- assert** — print help on assert options
- ea[:<packagename>...|:<classname>]** *or*
-enableassertions[:<packagename>...|:<classname>] — enable assertions
- da[:<packagename>...|:<classname>]** *or*
-disableassertions[:<packagename>...|:<classname>] — disable assertions
- esa** *or* **-enablesystemassertions** — enable system assertions
- dsa** *or* **-disablesystemassertions** — disable system assertions

D.2 Script options

- t** *or* **--trace** — Print the command being executed
- cp** *or* **-classpath** *<PATH>* — Append *<PATH>* to the java classpath
- P** *or* **-profile** *or* **--profile** — Run with profiling
- J<arg>** — Pass *<arg>* to java, e.g., use **-J-verbose** to make java execution verbose
- V** *or* **--version** — Same as **-version**
- v** *or* **--verbose** — Same as **-verbose**
- h** *or* **--help** — Same as **-help**

E. APACHE ANT INTEGRATION

XJ programs can also be compiled from Apache Ant [[Apache Software Foundation 2005](#)] **build.xml** scripts by using the provided `<xjc>` task. Due to a bug in earlier versions of Ant, the `<xjc>` task requires Ant version 1.6.5 or later. Just as `xjc` compiles `.java` files, the `<xjc>` task can be used as a replacement for the `<javac>` task (except that it also compiles `.xj` programs). Below is a sample invocation of the `<xjc>` task to compile all files in a given directory:

```
<xjc destdir="${build}" debug="on" verbose="on"
    includes="${src}/**/*.java,${src}/**/*.xj">
  <src location="${src}"/>
  <classpath refid="standard.classpath"/>
</xjc>
```

Before the `<xjc>` task can be used, it needs to be defined. This is normally done by including the provided **antlib** file. Below is an example **build.xml** target, `<declarexjctask>` that defines the `<xjc>` task:

```
<target name="declarexjctask">
  <taskdef resource="com/ibm/xj/antlib.xml" onerror="fail"
    classpathref="standard.classpath"/>
  <property name="xjc.declared" value="true"/>
</target>
```

The **antlib** file is included in the distribution as part of the `xj-ant.jar` file, which needs to be in the classpath. Any target that uses the `<xjc>` task needs to depend on the `<declarexjctask>` target above. For convenience, upon successful declaration of the `<xjc>` task, the above target also defines the `xjc.declared` property, which the dependent targets could be made conditional upon:

```
<path id="standard.classpath">
  <pathelement location="${build}"/>
  ...
  <fileset dir="${xjdir}/lib">
    <include name="xj-ant.jar"/>
    <include name="xj-runtime.jar"/>
    <include name="xj-xpath.jar"/>
  </fileset>
</path>
...
<target name="compile" depends="init,declarexjctask" if="xjc.declared">
  <xjc ...>...</xjc>
  ...
</target>
```

The XJ distribution includes a sample **build.xml** file in the **samples** directory.

F. PLATFORM NOTES

The XJ compiler has been run on the following platforms using a Java 1.4.2 installation:

- Linux
- Cygwin
- Windows
- Mac OS X

Note that there may be problems with newer versions of Java. See Section 1.3 for instructions on filing bug reports.

The sections below contain platform-specific notes.

F.1 Linux

Linux Java installations usually have `$JAVA_HOME` already set correctly via a `/etc/profile.d` script.

F.2 Cygwin

Cygwin is unique among the platforms in that it doesn't have a native Java package, and has to use the one from Windows. One consequence of that is that Java programs won't understand Cygwin paths, and conversion to Windows paths has to be made. Relative paths usually work best on Cygwin.

When both a script and an executable with the same name are present, Cygwin will prefer the executable. This allows the use of the Windows executables, `xjc.exe` and `xj.exe` on Cygwin, but also means that `$CLASSPATH`, `$JAVA_HOME`, *etc.*, need to be set in the Windows format (with drive letters and ';' as the separator, for a looser integration with the rest of the tools).

It is not possible to achieve full integration, as XJ programs will still require data file names and other parameters passed via the command line, properties, or config files as Windows paths. The regression test scripts are patched to work around this issue, but again, it is best to use relative paths with XJ on Cygwin. Note also that, as in any Unix shell, if backslashes are used on the command line in Cygwin, they need to be either doubled, or properly quoted using single quotes (''). To avoid the need for quoting, forward slashes ('/') can also be used for Windows style paths.

F.3 Windows

The regression test framework requires Cygwin to run under Windows, and will not work in a pure Windows environment.

F.4 Mac OS X

XJ should work on Mac OS X. However, it hasn't been extensively tested on this platform.

G. XJ DISTRIBUTION DETAILS

As mentioned in Section 2.1, the XJ distribution consists of two `.zip` archives. Description of the contents of the archives is provided below.

xj-bin-*<version>*.zip

The XJ binary distribution. This distribution contains all the files needed to compile and run XJ programs on the command line, some sample XJ programs, and the PDF version of this manual. This is the only archive you need to compile and run XJ programs. The directory structure is as follows:

```
XJJD/      XJ installation main directory
|- bin/     Scripts for compiling and running XJ programs
|- doc/     XJ documentation (this file)
|- lib/     The XJ compiler and runtime JARs
|- licenses/ Licenses governing the XJ distribution
|- regression/ The XJ regression testing framework
|- samples/ Sample XJ programs and build script
  \- share/ External JARs that XJ uses
```

xj-src-*<version>*.zip

The modified source files from the Eclipse JDT that the XJ compiler is based on. This file does not need to be downloaded to run XJ. The directory structure is as follows:

```
XJJD/      XJ installation main directory
  \- src/   Source directory
      |- batch/ Modified JDT batch runner sources
      \- compiler/ Modified JDT compiler sources
```

Note: These files are released as required by, and under the terms of the CPL.

The `<version>` can also be in the form of `<date>-<time>`.