Definitive XML Schema

Second Edition

The Charles F. Goldfarb Definitive XML Series

Priscilla Walmsley
- Definitive XML Schema Second Edition

Charles F. Goldfarb and Paul Prescod
- Charles F. Goldfarb's XML Handbook™ Fifth Edition

Rick Jelliffe
- The XML and SGML Cookbook: Recipes for Structured Information

Charles F. Goldfarb, Steve Pepper, and Chet Ensign
- SGML Buyer's Guide: Choosing the Right XML and SGML Products and Services

G. Ken Holman
- Definitive XSL-FO
- Definitive XSLT and XPath

Bob DuCharme
- XML: The Annotated Specification
- SGML CD

Truly Donovan
- Industrial-Strength SGML: An Introduction to Enterprise Publishing

Lars Marius Garshol
- Definitive XML Application Development

JP Morgenthal with Bill la Forge
- Enterprise Application Integration with XML and Java

Michael Leventhal, David Lewis, and Matthew Fuchs
- Designing XML Internet Applications

Adam Hocek and David Cuddihy
- Definitive VoiceXML

Dmitry Kirsanov
- XSLT 2.0 Web Development

Yuri Rubinsky and Murray Maloney
- SGML on the Web: Small Steps Beyond HTML

David Megginson
- Structuring XML Documents

Sean McGrath
- XML Processing with Python
- XML by Example: Building E-commerce Applications
- ParseMe.1st: SGML for Software Developers

Chet Ensign
- $GML: The Billion Dollar Secret

Ron Turner, Tim Douglass, and Audrey Turner
- ReadMe.1st: SGML for Writers and Editors

Charles F. Goldfarb and Priscilla Walmsley
- XML in Office 2003: Information Sharing with Desktop XML

Michael Floyd
- Building Web Sites with XML

Fredrick Thomas Martin
- TOP SECRET Intranet: How U.S. Intelligence Built Intelink—The World's Largest, Most Secure Network

J. Craig Cleaveland
- Program Generators with XML and Java

About the Series Author

Charles F. Goldfarb is the father of XML technology. He invented SGML, the Standard Generalized Markup Language on which both XML and HTML are based. You can find him on the Web at: www.xmlbooks.com.

About the Series Logo

The rebus is an ancient literary tradition, dating from 16th century Picardy, and is especially appropriate to a series involving fine distinctions between markup and text, metadata and data. The logo is a rebus incorporating the series name within a stylized XML comment declaration.

Definitive XML Schema

Second Edition

Priscilla Walmsley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Titles in this series are produced using XML, SGML, and/or XSL. XSL-FO documents are rendered into PDF by the XEP Rendering Engine from RenderX: www.renderx.com.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382–3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com

Library of Congress Cataloging-in-Publication Data is on file

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236–3290.

ISBN-13: 978-0-132-88672-7
ISBN-10: 0-132-88672-3
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, MI.
First printing: September 2012

Editor-in-Chief: Mark L. Taub
Managing Editor: Kristy Hart
Book Packager: Alina Kirsanova
Cover Designer: Alan Clements

To Doug, my SH

This page intentionally left blank

Overview

Contents

Foreword

clas·sic *(adjective)*
judged over a period of time to be important and of the highest quality:

a classic novel
a classic car

Neither this definition, nor any of the leading dictionary definitions, has a usage example anything like:

a classic work on high-tech software

After all, it is a rare book on software that even survives long enough to be "judged over a period of time."

Nevertheless, *Definitive XML Schema* satisfies every definition of "classic." It is one of the elite few software books that have been in print continuously for over ten years, and an essential trustworthy guide for tens of thousands of readers.

This *Second Edition* continues to be an essential and trustworthy classic:

Essential because in the last ten years XML has become the accepted standard for data interchange, and XML Schema 1.0 is largely responsible. Now version 1.1 has extended the ability to specify and validate document data, to a degree previously possible only for databases. These updates are covered in this book by extensive revisions—the most significant 250 of which are flagged in the text and table of contents. Hundreds more unflagged revisions reflect W3C corrections of XML Schema errata, and ten years of evolving "best practices."

Trustworthy because it is both authoritative and accurate.

- The **author(ity)**, Priscilla Walmsley, is a noted consultant who has been using XML Schema ever since she helped develop it as a member of the W3C XML Schema Group. She personally devised many of the current "best practices" described in this book. Priscilla is the Editor of the W3C XML Schema Primer, Second Edition.
- **Accuracy** was preserved by using the same XML-based production system that was used in 2002, operated by the same team of XML experts who read and thoroughly understood the book. Priscilla's original XML source (in which she had personally tagged the version 1.1 revisions) was used throughout production. Dmitry Kirsanov copy-edited and proofed it, while Alina Kirsanova prepared the index, coded the XSL transformations, and generated the camera-ready PDFs.

The result, as you will see, retains the structure, clarity, patient explanations, validated examples (over 450!), and well-reasoned advice that critics praised in the 2002 edition—but now they are ten years more up-to-date.

And after you've read *Definitive XML Schema, Second Edition*, it won't take another ten years for you, too, to judge it a classic.

Charles F. Goldfarb
Belmont, CA
August 2012

Acknowledgments

First and foremost, I would like to thank Charles Goldfarb for his invaluable guidance and support. Alina Kirsanova and Dmitry Kirsanov did an excellent job preparing this book for publication. I would also like to thank Mark Taub at Prentice Hall for his hand in the making this work possible.

Of course, this book would not have been possible without the efforts of all of the members of the W3C XML Schema Working Group, with whom I have had the pleasure of working for six years. The content of this book was shaped by the questions and comments of the people who contribute to XML-DEV and xmlschema-dev.

Finally, I'd like to thank my Dad for teaching me to "get stuck into it," a skill which allowed me to complete this substantial project.

Priscilla Walmsley
Traverse City, Michigan
March 2012

How to use this book

This book covers the two versions of XML Schema—1.0 and 1.1—and provides revision bars to assist access to just one or the other. In referring to both versions as "XML Schema," the book follows customary practice, despite the official name of 1.1 being "W3C XML Schema Definition Language (XSD) 1.1." For either version, the book is useable as both a tutorial and a reference.

As a tutorial, the book can be read from cover to cover with confidence that each topic builds logically on the information that was previously covered. (Of course, knowledge of XML itself is always a prerequisite to learning about XML Schema, and is assumed in this book.)

When using this book as a reference, you have several access options available to you:

- A comprehensive index starts on page 699.
- An alphabetical list of all the elements and attributes that make up the XML Schema syntax is in Appendix A on p. 648. For each there is a reference to further coverage in the body of the book.
- XML Schema includes a basic set of datatypes, known formally as the "built-in simple types." They are listed in Appendix B on p. 690. This appendix also refers to more detailed descriptions in the body of the book.

- The major changes in version 1.1 of XML Schema are summarized in Section 23.5.1 on p. 640, with references to detailed coverage elsewhere in the book.

Revisions in the Second Edition

This edition of *Definitive XML Schema* contains more than 500 revisions, covering such new and updated topics as:

- W3C published corrections for errata in XML Schema 1.0
- Current "best practices" after ten years of experience
- XML information modeling for relational and object-oriented modeling paradigms
- Schema design: evaluating pros and cons of alternatives
- Schema strategy: formulating a coherent strategy for schema development and maintenance
- Version 1.1 updates and additions to XML Schema

Identifying 1.1-related revisions

The author has chosen a "1.1 subset" of the book revisions, comprising the 250 most significant revisions that deal with version 1.1. If a section, table, example, or paragraph has content entirely from the "1.1 subset," there is a solid gray revision bar on its right. If other material might be included, the bar is a gray dotted line.

Strategies for using the revision bars

If your interest is **solely 1.0** (perhaps because your software does not yet support 1.1), you may decide to focus on content that either has a dotted revision bar or no bar at all.

If you are interested only in what is **new in 1.1** (presumably because you already know 1.0), consider content having either a solid or dotted revision bar in deciding where to focus your reading.

Finally, if your interest is **all of 1.1** (because you don't already know 1.0), you can easily disregard the revision bars (that's why they are grayed out ☺).

Syntax tables

This book contains syntax tables, each summarizing the allowed syntax of an XML Schema component. The first such table does not occur until Section 4.2 on p. 58, by which point the undefined terms in this explanation will have been introduced.

Syntax tables, whose captions all start with "XSD Syntax," look like the example below, which shows the syntax for named simple types. It contains the following information:

- The element name(s) used for this XML Schema component.
- The possible parent element name(s). Note that "1.1", printed white on a gray box, precedes `override` to identify it as a construct that is only permitted in version 1.1. This convention is followed in all syntax tables; it occurs once more in this table.
- A list of allowed attributes, along with their types, valid values, and brief descriptions. The names of required attributes appear in bold font. Default values appear in italics in the ***Type*** column.
- The allowed child elements, shown as a content model that uses, for compactness, the XML DTD syntax. Commas indicate that child elements must appear in the order shown, while vertical bars (|) indicate a choice among child elements. Occurrence constraints indicate how many of each may appear: `?` means zero or one, `*` means zero or more, and `+` means one or more. Otherwise, one and only one is required. In this example, the allowed content is zero or one `annotation` element, followed by a choice of either one `restriction`, one `list`, or one `union` element.

Table XSD Syntax: named simple type definition

Name		
simpleType		
Parents		
schema, redefine, override		
Attribute name	***Type***	***Description***
id	ID	Unique ID.
name	NCName	Simple type name.
final	"#all" | list of ("restriction" | "list" | "union" | "extension")	Whether other types can be derived from this one.
Content		
annotation?, (restriction | list | union)		

In some cases, there is more than one syntax table for the same element name, because certain element names in XML Schema have multiple uses. For example, `simpleType` is used for both named simple types and anonymous simple types. Each of these use cases of `simpleType` allows different attributes and a different set of parent elements, so each is described with its own table.

Companion website

This book has a companion website, maintained by the author, at www.datypic.com/books/defxmlschema2. On the website, you can view any errata and download the examples from this book. In addition to the examples that appear in the book, which are generally concise in order to illustrate a particular point, the website also has larger, more comprehensive instances and schemas that can be copied or used to test validation.

Simple types

Chapter 8

Both element and attribute declarations can use simple types to describe their data content. This chapter introduces simple types and explains how to define your own atomic simple types for use in your schemas.

8.1 | Simple type varieties

There are three varieties of simple types: atomic types, list types, and union types.

1. *Atomic types* have values that are indivisible, such as `10` or `large`.
2. *List types* have values that are whitespace-separated lists of atomic values, such as `<availableSizes>10 large 2</availableSizes>`.
3. *Union types* may have values that are either atomic values or list values. What differentiates them is that the set of valid values, or "value space," for the type is the union of the value spaces of

two or more other simple types. For example, to represent a dress size, you may define a union type that allows a value to be either an integer from `2` through `18`, or one of the string values `small`, `medium`, or `large`.

List and union types are covered in Chapter 10.

8.1.1 *Design hint: How much should I break down my data values?*

Data values should be broken down to the most atomic level possible. This allows them to be processed in a variety of ways for different uses, such as display, mathematical operations, and validation. It is much easier to concatenate two data values back together than it is to split them apart. In addition, more granular data is easier to validate.

It is a fairly common practice to put a data value and its units in the same element, for example `<length>3cm</length>`. However, the preferred approach is to have a separate data value, preferably an attribute, for the units, for example `<length units="cm">3</length>`.

Using a single concatenated value is limiting because:

- It is extremely cumbersome to validate. You have to apply a complicated pattern that would need to change every time a unit type is added.
- You cannot perform comparisons, conversions, or mathematical operations on the data without splitting it apart.
- If you want to display the data item differently (for example, as “3 centimeters” or “3 cm” or just “3”, you have to split it apart. This complicates the stylesheets and applications that process instance documents.

It is possible to go too far, though. For example, you may break a date down as follows:

```
<orderDate>
  <year>2001</year>
  <month>06</month>
  <day>15</day>
</orderDate>
```

This is probably overkill unless you have a special need to process these items separately.

8.2 | Simple type definitions

8.2.1 *Named simple types*

Simple types can be either named or anonymous. Named simple types are always defined globally (i.e., their parent is always `schema`[1]) and are required to have a name that is unique among the types (both simple and complex) in the schema. The syntax for a named simple type definition is shown in Table 8–1.

The `name` of a simple type must be an XML non-colonized name, which means that it must start with a letter or underscore, and may only contain letters, digits, underscores, hyphens, and periods. You cannot include a namespace prefix when defining the type; it takes its namespace from the target namespace of the schema document.

All examples of named types in this book have the word "Type" at the end of their names to clearly distinguish them from element and attribute names. However, this is a convention and not a requirement. You can even have a type definition and an element declaration using the same name, but this is not recommended because it can be confusing.

Example 8–1 shows the definition of a named simple type `DressSizeType` along with an element declaration that references it. Named types can be used in multiple element and attribute declarations.

---

1. Except in the case of a redefine or override.

Table 8–1 XSD Syntax: named simple type definition

Name		
simpleType		
Parents		
schema, redefine, 1.1 override		
Attribute name	*Type*	*Description*
id	ID	Unique ID.
name	NCName	Simple type name.
final	"#all" | list of ("restriction" | "list" | "union" | 1.1 "extension")	Whether other types can be derived from this one (see Section 8.5); defaults to finalDefault of schema.
Content		
annotation?, (restriction | list | union)		

Example 8–1. Defining and referencing a named simple type

```
<xs:simpleType name="DressSizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="2"/>
    <xs:maxInclusive value="18"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="size" type="DressSizeType"/>
```

8.2.2 *Anonymous simple types*

Anonymous types, on the other hand, must not have names. They are always defined entirely within an element or attribute declaration, and may only be used once, by that declaration. Defining a type anonymously prevents it from ever being restricted, used in a list or union, redefined, or overridden. The syntax to define an anonymous simple type is shown in Table 8–2.

Table 8–2 XSD Syntax: anonymous simple type definition

Name		
`simpleType`		
Parents		
`element`, `attribute`, `restriction`, `list`, `union`, 1.1 `alternative`		
Attribute name	***Type***	***Description***
`id`	`ID`	Unique ID.
Content		
`annotation?, (restriction | list | union)`		

Example 8–2 shows the definition of an anonymous simple type within an element declaration.

Example 8–2. Defining an anonymous simple type

```
<xs:element name="size">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="2"/>
      <xs:maxInclusive value="18"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

8.2.3 *Design hint: Should I use named or anonymous types?*

The advantage of named types is that they may be defined once and used many times. For example, you may define a type named

ProductCodeType that lists all of the valid product codes in your organization. This type can then be used in many element and attribute declarations in many schemas. This has the advantages of

- Encouraging consistency throughout the organization
- Reducing the possibility of error
- Requiring less time to define new schemas
- Simplifying maintenance, because new product codes need only be added in one place

If a type is named, you can also derive new types from it, which is another way to promote reuse and consistency.

Named types can also make a schema more readable when its type definitions are complicated.

An anonymous type, on the other hand, can be used only in the element or attribute declaration that contains it. It can never be redefined, overridden, have types derived from it, or be used in a list or union type. This can seriously limit its reusability, extensibility, and ability to change over time.

However, there are cases where anonymous types are preferable to named types. If the type is unlikely to ever be reused, the advantages listed above no longer apply. Also, there is such a thing as too much reuse. For example, if an element can contain the values 1 through 10, it does not make sense to define a type named OneToTenType to be reused by other unrelated element declarations with the same value space. If the value space for one of the element declarations using that named type changes but the other element declarations stay the same, it actually makes maintenance more difficult, because a new type would need to be defined at that time.

In addition, anonymous types can be more readable when they are relatively simple. It is sometimes desirable to have the definition of the type right there with the element or attribute declaration.

8.3 | Simple type restrictions

Every simple type is a restriction of another simple type, known as its base type. It is not possible to extend a simple type, except by adding attributes which results in a complex type. This is described in Section 13.4.1 on p. 306.

Every new simple type restricts the value space of its base type in some way. Example 8–3 shows a definition of `DressSizeType` that restricts the built-in type `integer`.

Example 8–3. **Deriving a simple type from a built-in simple type**

```
<xs:simpleType name="DressSizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="2"/>
    <xs:maxInclusive value="18"/>
    <xs:pattern value="\d{1,2}"/>
  </xs:restriction>
</xs:simpleType>
```

Simple types may also restrict user-derived simple types that are defined in the same schema document, or even in a different schema document. For example, you could further restrict `DressSizeType` by defining another simple type, `MediumDressSizeType`, as shown in Example 8–4.

A simple type restricts its base type by applying facets to restrict its values. In Example 8–4, the facets `minInclusive` and `maxInclusive` are used to restrict the value of `MediumDressSizeType` to be between `8` and `12` inclusive.

Example 8–4. **Deriving a simple type from a user-derived simple type**

```
<xs:simpleType name="MediumDressSizeType">
  <xs:restriction base="DressSizeType">
    <xs:minInclusive value="8"/>
    <xs:maxInclusive value="12"/>
  </xs:restriction>
</xs:simpleType>
```

8.3.1 *Defining a restriction*

The syntax for a `restriction` element is shown in Table 8–3. You must specify one base type either by using the `base` attribute or by defining the simple type anonymously using a `simpleType` child. The option of using a `simpleType` child is generally only useful when restricting list types, as described in Section 10.3.3 on p. 190.

Table 8–3 XSD Syntax: simple type restriction

Name		
`restriction`		
Parents		
`simpleType`		
Attribute name	***Type***	***Description***
`id`	`ID`	Unique ID.
`base`	`QName`	Simple type that is being restricted; either a `base` attribute or a `simpleType` child is required.
Content		
`annotation?, simpleType?, (minExclusive | minInclusive | maxExclusive | maxInclusive | length | minLength | maxLength | totalDigits | fractionDigits | enumeration | pattern | whiteSpace | 1.1 assertion | 1.1 explicitTimezone |` 1.1 *{any element in another namespace}*`)*`		

Within a `restriction` element, you can specify any of the facets, in any order. However, the only facets that may appear more than once in the same restriction are `pattern`, `enumeration`, and `assertion`. It is legal to define a restriction that has no facets specified. In this case, the derived type allows the same values as the base type.

8.3.2 *Overview of the facets*

The available facets are listed in Table 8–4.

Table 8–4 **Facets**

Facet	*Meaning*
`minExclusive`	Value must be greater than *x*.
`minInclusive`	Value must be greater than or equal to *x*.
`maxInclusive`	Value must be less than or equal to *x*.
`maxExclusive`	Value must be less than *x*.
`length`	The length of the value must be equal to *x*.
`minLength`	The length of the value must be greater than or equal to *x*.
`maxLength`	The length of the value must be less than or equal to *x*.
`totalDigits`	The number of significant digits must be less than or equal to *x*.
`fractionDigits`	The number of fractional digits must be less than or equal to *x*.
`whiteSpace`	The schema processor should either preserve, replace, or collapse whitespace depending on *x*.
`enumeration`	*x* is one of the valid values.
`pattern`	*x* is one of the regular expressions that the value may match.
1.1 `explicitTimezone`	The time zone part of the date/time value is required, optional, or prohibited depending on *x*.
1.1 `assertion`	The value must conform to a constraint in the XPath expression.

The syntax for applying a facet is shown in Table 8–5. All facets (except `assertion`) must have a `value` attribute, which has different valid values depending on the facet. Most facets may also have a `fixed` attribute, as described in Section 8.3.4 on p. 140.

Table 8–5 XSD Syntax: facet

Name		
`minExclusive`, `minInclusive`, `maxExclusive`, `maxInclusive`, `length`, `minLength`, `maxLength`, `totalDigits`, `fractionDigits`, `enumeration`, `pattern`, `whiteSpace`, 1.1 `explicitTimezone`†		
Parents		
`restriction`		
Attribute name	*Type*	*Description*
`id`	`ID`	Unique ID.
`value`	various	Value of the restricting facet.
`fixed`	`boolean`: *`false`*	Whether the facet is fixed and therefore cannot be restricted further (see Section 8.3.4); not applicable for `pattern`, `enumeration`.
Content		
`annotation?`		
† The `assertion` facet has a different syntax that is described in Table 14–1.		

Certain facets are not applicable to some types. For example, it does not make sense to apply the `fractionDigits` facet to a character string type. There is a defined set of applicable facets for each of the built-in types.[1] If a facet is applicable to a built-in type, it is also applicable to atomic types that are derived from it. For example, since the `length` facet is applicable to `string`, if you derive a new type from

1. Technically, it is the primitive types that have applicable facets, with the rest of the built-in types inheriting that applicability from their base types. However, since most people do not have the built-in type hierarchy memorized, it is easier to list applicable facets for all the built-in types.

string, the length facet is also applicable to your new type. Section 8.4 on p. 142 describes each of the facets in detail and lists the built-in types to which the facet can apply.

8.3.3 *Inheriting and restricting facets*

When a simple type restricts its base type, it inherits all of the facets of its base type, its base type's base type, and so on back through its ancestors. Example 8–4 showed a simple type MediumDressSizeType whose base type is DressSizeType. DressSizeType has a pattern facet which restricts its value space to one- or two-digit numbers. Since MediumDressSizeType inherits all of the facets from DressSizeType, this same pattern facet applies to MediumDressSizeType also. Example 8–5 shows an equivalent definition of MediumDressSizeType where it restricts integer and has the pattern facet applied.

Example 8–5. **Effective definition of** MediumDressSizeType

```
<xs:simpleType name="MediumDressSizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="8"/>
    <xs:maxInclusive value="12"/>
    <xs:pattern value="\d{1,2}"/>
  </xs:restriction>
</xs:simpleType>
```

Sometimes a simple type definition will include facets that are also specified for one of its ancestors. In Example 8–4, MediumDressSizeType includes minInclusive and maxInclusive, which are also applied to its base type, DressSizeType. The minInclusive and maxInclusive facets of MediumDressSizeType (whose values are 8 and 12, respectively) override those of DressSizeType (2 and 18, respectively).

It is a requirement that the facets of a derived type (in this case MediumDressSizeType) be more restrictive than those of the base type. In Example 8–6, we define a new restriction of DressSizeType, called `SmallDressSizeType`, and set `minInclusive` to `0`. This type definition is illegal, because it attempts to expand the value space by allowing `0`, which was not valid for `DressSizeType`.

Example 8–6. **Illegal attempt to extend a simple type**

```
<xs:simpleType name="SmallDressSizeType">
  <xs:restriction base="DressSizeType">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="6"/>
  </xs:restriction>
</xs:simpleType>
```

This rule also applies when you are restricting the built-in types. For example, the `short` type has a `maxInclusive` value of `32767`. It is illegal to define a restriction of `short` that sets `maxInclusive` to `32768`.

Although `enumeration` facets can appear multiple times in the same type definition, they are treated in much the same way. If both a derived type and its ancestor have a set of `enumeration` facets, the values of the derived type must be a subset of the values of the ancestor. An example of this is provided in Section 8.4.4 on p. 145.

Likewise, the `pattern` facets specified in a derived type must allow a subset of the values allowed by the ancestor types. A schema processor will not necessarily check that the regular expressions represent a subset; instead, it will validate instances against the patterns of both the derived type and all the ancestor types, effectively taking the intersection of the pattern values.

8.3.4 *Fixed facets*

When you define a simple type, you can fix one or more of the facets. This means that further restrictions of this type cannot change the value of the facet. Any of the facets may be fixed, with the exception of `pattern`, `enumeration`, and `assertion`. Example 8–7 shows our

DressSizeType with fixed minInclusive and maxInclusive facets, as indicated by a fixed attribute set to true.

Example 8–7. **Fixed facets**

```
<xs:simpleType name="DressSizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="2" fixed="true"/>
    <xs:maxInclusive value="18" fixed="true"/>
    <xs:pattern value="\d{1,2}"/>
  </xs:restriction>
</xs:simpleType>
```

With this definition of DressSizeType, it would have been illegal to define the MediumDressSizeType as shown in Example 8–4 because it attempts to override the minInclusive and maxInclusive facets which are now fixed. Some of the built-in types have fixed facets that cannot be overridden. For example, the built-in type integer has its fractionDigits facet fixed at 0, so it is illegal to derive a type from integer and specify a fractionDigits that is not 0.

8.3.4.1 Design hint: When should I fix a facet?

Fixing facets makes your type less flexible and discourages other schema authors from reusing it. Keep in mind that any types that may be derived from your type must be more restrictive, so you are not at risk that your type will be dramatically changed if its facets are unfixed.

A justification for fixing facets might be that changing that facet value would significantly alter the meaning of the type. For example, suppose you want to define a simple type that represents price. You define a Price type and fix the fractionDigits at 2. This still allows other schema authors to restrict Price to define other types, for example, by limiting it to a certain range of values. However, they cannot modify the fractionDigits of the type, because this would result in a type not representing a price in dollars and cents.

8.4 | Facets

8.4.1 *Bounds facets*

The four bounds facets (`minInclusive`, `maxInclusive`, `minExclusive`, and `maxExclusive`) restrict a value to a specified range. Our previous examples applied `minInclusive` and `maxInclusive` to restrict the value space of `DressSizeType`. While `minInclusive` and `maxInclusive` specify boundary values that are included in the valid range, `minExclusive` and `maxExclusive` specify bounds that are excluded from the valid range.

There are several constraints associated with the bounds facets:

- `minInclusive` and `minExclusive` cannot both be applied to the same type. Likewise, `maxInclusive` and `maxExclusive` cannot both be applied to the same type. You may, however, mix and match, applying, for example, `minInclusive` and `maxExclusive` together. You may also apply just one end of the range, such as `minInclusive` only.
- The value for the lower bound (`minInclusive` or `minExclusive`) must be less than or equal to the value for the upper bound (`maxInclusive` or `maxExclusive`).
- The facet value must be a valid value for the base type. For example, when restricting `integer`, it is illegal to specify a `maxInclusive` value of `18.5`, because `18.5` is not a valid integer.

The four bounds facets can be applied only to the date/time and numeric types, and the types derived from them. Special consideration should be given to time zones when applying bounds facets to date/time types. For more information, see Section 11.4.15 on p. 235.

8.4.2 *Length facets*

The `length` facet allows you to limit values to a specific length. If it is a string-based type, length is measured in number of characters. This includes the XML DTD types and `anyURI`. If it is a binary type, length is measured in octets of binary data. If it is a list type, length is measured as the number of items in the list. The facet value for `length` must be a nonnegative integer.

The `minLength` and `maxLength` facets allow you to limit a value's length to a specific range. Either of both of these facets may be applied. If they are both applied, `minLength` must be less than or equal to `maxLength`. If the `length` facet is applied, neither `minLength` nor `maxLength` may be applied. The facet values for `minLength` and `maxLength` must be nonnegative integers.

The three length facets (`length`, `minLength`, `maxLength`) can be applied to any string-based types (including the XML DTD types), the binary types, and `anyURI`. They cannot be applied to the date/time types, numeric types, or `boolean`.

8.4.2.1 Design hint: What if I want to allow empty values?

Many of the built-in types do not allow empty values. Types other than `string`, `normalizedString`, `token`, `hexBinary`, `base64Binary`, and `anyURI` do not allow empty values unless `xsi:nil` appears in the element tag.

You may have an integer that you want to be either between `2` and `18`, or empty. First, consider whether you want to make the element (or attribute) optional. In this case, if the data is absent, the element will not appear at all. However, sometimes it is desirable for the element to appear, as a placeholder, or perhaps it is unavoidable because of the technology used to generate the instance.

If you do determine that the elements must be able to appear empty, you must define a union type that includes both the integer type and an empty string, as shown in Example 8–8.

Example 8–8. Union allowing an empty value

```
<xs:simpleType name="DressSizeType">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="2"/>
        <xs:maxInclusive value="18"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value=""/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

8.4.2.2 Design hint: What if I want to restrict the length of an integer?

The `length` facet only applies to the string-based types, the XML DTD types, the binary types, and `anyURI`. It does not make sense to try to limit the length of the date/time types because they have fixed lexical representations. But what if you want to restrict the length of an integer value?

You can restrict the lower and upper bounds of an integer by applying bounds facets, as discussed in Section 8.4.1 on p. 142. You can also control the number of significant digits in an integer using the `totalDigits` facet, as discussed in Section 8.4.3 on p. 145. However, these facets do not consider leading zeros as significant. Therefore, they cannot force an integer to appear in the instance with a specific number of digits. To do this, you need a pattern. For example, the pattern `\d{1,2}` used in our `DressSizeType` example forces the size to be one or two digits long, so `012` would be invalid.

Before taking this approach, however, you should reconsider whether it is really an integer or a string. See Section 11.3.3.1 on p. 220 for a discussion of this issue.

8.4.3 `totalDigits` *and* `fractionDigits`

The `totalDigits` facet allows you to specify the maximum number of digits in a number. The facet value for `totalDigits` must be a positive integer.

The `fractionDigits` facet allows you to specify the maximum number of digits in the fractional part of a number. The facet value for `fractionDigits` must be a nonnegative integer, and it must not exceed the value for `totalDigits`, if one exists.

The `totalDigits` facet can be applied to `decimal` or any of the integer types, as well as types derived from them. The `fractionDigits` facet may only be applied to `decimal`, because it is fixed at `0` for all integer types.

8.4.4 *Enumeration*

The `enumeration` facet allows you to specify a distinct set of valid values for a type. Unlike most other facets (except `pattern` and `assertion`), the `enumeration` facet can appear multiple times in a single restriction. Each enumerated value must be unique, and must be valid for that type. If it is a string-based or binary type, you may also specify the empty string in an enumeration value, which allows elements or attributes of that type to have empty values.

Example 8–9 shows a simple type `SMLXSizeType` that allows the values `small`, `medium`, `large`, and `extra large`.

Example 8–9. Applying the enumeration facet

```
<xs:simpleType name="SMLXSizeType">
  <xs:restriction base="xs:token">
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
    <xs:enumeration value="extra large"/>
  </xs:restriction>
</xs:simpleType>
```

When restricting types that have enumerations, it is important to note that you must *restrict*, rather than *extend*, the set of enumeration values. For example, if you want to restrict the valid values of `SMLSizeType` to only be `small`, `medium`, and `large`, you could define a simple type as in Example 8–10.

Example 8–10. Restricting an enumeration

```
<xs:simpleType name="SMLSizeType">
  <xs:restriction base="SMLXSizeType">
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
  </xs:restriction>
</xs:simpleType>
```

Note that you need to repeat all of the enumeration values that apply to the new type. This example is legal because the values for `SMLSizeType` (`small`, `medium`, and `large`) are a subset of the values for `SMLXSizeType`. By contrast, Example 8–11 attempts to add an `enumeration` facet to allow the value `extra small`. This type definition is illegal because it attempts to extend rather than restrict the value space of `SMLXSizeType`.

Example 8–11. Illegal attempt to extend an enumeration

```
<xs:simpleType name="XSMLXSizeType">
  <xs:restriction base="SMLXSizeType">
    <xs:enumeration value="extra small"/>
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
    <xs:enumeration value="extra large"/>
  </xs:restriction>
</xs:simpleType>
```

The only way to add an enumeration value to a type is by defining a union type. Example 8–12 shows a union type that adds the value

Example 8–12. Using a union to extend an enumeration

```
<xs:simpleType name="XSMLXSizeType">
  <xs:union memberTypes="SMLXSizeType">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="extra small"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

`extra small` to the set of valid values. Union types are described in detail in Section 10.2 on p. 183.

When enumerating numbers, it is important to remember that the `enumeration` facet works on the actual value of the number, not its lexical representation as it appears in an XML instance. Example 8–13 shows a simple type `NewSmallDressSizeType` that is based on `integer`, and specifies an enumeration of `2`, `4`, and `6`. The two instance elements shown, which contain `2` and `02`, are both valid. This is because `02` is equivalent to `2` for integer-based types. However, if the base type of `NewSmallDressSizeType` had been `string`, the

Example 8–13. Enumerating numeric values

Schema:

```
<xs:simpleType name="NewSmallDressSizeType">
  <xs:restriction base="xs:integer">
    <xs:enumeration value="2"/>
    <xs:enumeration value="4"/>
    <xs:enumeration value="6"/>
  </xs:restriction>
</xs:simpleType>
```

Valid instances:

```
<size>2</size>
<size>02</size>
```

value `02` would not be valid, because the strings `2` and `02` are not the same. If you wish to constrain the lexical representation of a numeric type, you should apply the `pattern` facet instead. For more information on type equality in XML Schema, see Section 11.7 on p. 253.

The `enumeration` facet can be applied to any type except `boolean`.

8.4.5 *Pattern*

The `pattern` facet allows you to restrict values to a particular pattern, represented by a regular expression. Chapter 9 provides more detail on the rules for the regular expression syntax. Unlike most other facets (except `enumeration` and `assertion`), the `pattern` facet can be specified multiple times in a single restriction. If multiple `pattern` facets are specified in the same restriction, the instance value must match at least one of the patterns. It is not required to match all of the patterns.

Example 8–14 shows a simple type `DressSizeType` that includes the pattern `\d{1,2}`, which restricts the size to one or two digits.

Example 8–14. Applying the pattern facet

```
<xs:simpleType name="DressSizeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="2"/>
    <xs:maxInclusive value="18"/>
    <xs:pattern value="\d{1,2}"/>
  </xs:restriction>
</xs:simpleType>
```

When restricting types that have patterns, it is important to note that you must *restrict*, rather than *extend*, the set of valid values that the patterns represent. In Example 8–15, we define a simple type `SmallDressSizeType` that is derived from `DressSizeType`, and add an additional `pattern` facet that restricts the size to one digit.

Example 8–15. Restricting a pattern

```
<xs:simpleType name="SmallDressSizeType">
  <xs:restriction base="DressSizeType">
    <xs:minInclusive value="2"/>
    <xs:maxInclusive value="6"/>
    <xs:pattern value="\d{1}"/>
  </xs:restriction>
</xs:simpleType>
```

It is not technically an error to apply a pattern facet that does not represent a subset of the ancestors' pattern facets. However, the schema processor tries to match the instance value against the pattern facets of both the type and its ancestors, ensuring that it is in fact a subset. Example 8–16 shows an illegal attempt to define a new size type that allows the size value to be up to three digits long. While the schema is not in error, it will not have the desired effect because the schema processor will check values against both the pattern of `LongerDressSizeType` and the pattern of `DressSizeType`. The value `004` would not be considered a valid instance of `LongerDressSizeType` because it does not conform to the pattern of `DressSizeType`.

Example 8–16. Illegal attempt to extend a pattern

```
<xs:simpleType name="LongerDressSizeType">
  <xs:restriction base="DressSizeType">
    <xs:pattern value="\d{1,3}"/>
  </xs:restriction>
</xs:simpleType>
```

Unlike the `enumeration` facet, the `pattern` facet applies to the lexical representation of the value. If the value `02` appears in an instance, the pattern is applied to the digits `02`, not `2` or `+2` or any other form of the integer.

The `pattern` facet can be applied to any type.

8.4.6 *Assertion*

The `assertion` facet allows you to specify additional constraints on values using XPath 2.0. Example 8–17 is a simple type with an assertion, namely that the value must be divisible by 2. It uses a facet named `assertion` with a `test` attribute that contains the XPath expression.

Simple type assertions are a flexible and powerful feature covered in more detail, along with complex type assertions, in Chapter 14.

Example 8–17. Simple type assertion

```
<xs:simpleType name="EvenDressSizeType">
  <xs:restriction base="DressSizeType">
    <xs:assertion test="$value mod 2 = 0" />
  </xs:restriction>
</xs:simpleType>
```

8.4.7 *Explicit Time Zone*

The `explicitTimezone` facet allows you to control the presence of an explicit time zone on a date/time value. Example 8–18 is a simple type based on `time` but with an explicit time zone required. The syntax of time zones is described in more detail in Section 11.4.13 on p. 233.

The `value` attribute of `explicitTimezone` has three possible values:

1. `optional`, making the time zone optional (the value for most built-in date/time types)
2. `required`, making the time zone required (the value for the `dateTimeStamp` built-in type)
3. `prohibited`, disallowing the time zone

Example 8–18. Explicit time zone

```
<xs:simpleType name="SpecificTimeType">
  <xs:restriction base="xs:time">
    <xs:explicitTimezone value="required"/>
  </xs:restriction>
</xs:simpleType>
```

8.4.8 *Whitespace*

The `whiteSpace` facet allows you to specify the whitespace normalization rules which apply to this value. Unlike the other facets, which restrict the value space of the type, the `whiteSpace` facet is an instruction to the schema processor on to what to do with whitespace. This type of facet is known as a *prelexical* facet because it results in some processing of the value before the other constraining facets are applied. The valid values for the `whiteSpace` facet are:

- `preserve`: All whitespace is preserved; the value is not changed.
- `replace`: Each occurrence of tab (`#x9`), line feed (`#xA`), and carriage return (`#xD`) is replaced with a single space (`#x20`).
- `collapse`: As with replace, each occurrence of tab (`#x9`), line feed (`#xA`), and carriage return (`#xD`) is replaced with a single space (`#x20`). After the replacement, all consecutive spaces are collapsed into a single space. In addition, leading and trailing spaces are deleted.

Table 8–6 shows examples of how values of a string-based type will be handled depending on its `whiteSpace` facet.

Table 8–6 Handling of string values depending on `whiteSpace` facet

Original string	`string` *(preserve)*	`normalizedString` *(replace)*	`token` *(collapse)*
`a string`	`a string`	`a string`	`a string`
`on` `two lines`	`on` `two lines`	`on  two lines`	`on two lines`
`has      spaces`	`has      spaces`	`has      spaces`	`has spaces`
`   leading tab`	`   leading tab`	`  leading tab`	`leading tab`
`    leading spaces`	`    leading spaces`	`    leading spaces`	`leading spaces`

The whitespace processing, if any, will happen first, before any validation takes place. In Example 8–9, the base type of `SMLXSizeType` is `token`, which has a `whiteSpace` facet of `collapse`. Example 8–19 shows valid instances of `SMLXSizeType`. They are valid because the leading and trailing spaces are removed, and the line feed is turned into a space. If the base type of `SMLXSizeType` had been `string`, the whitespace would have been left as is, and these values would have been invalid.

Example 8–19. Valid instances of `SMLXSizeType`

```
<size> small </size>

<size>extra
large</size>
```

Although you should understand what the `whiteSpace` facet represents, it is unlikely that you will ever apply it directly in your schemas. The `whiteSpace` facet is fixed at `collapse` for most built-in types. Only the string-based types can be restricted by a `whiteSpace` facet, but this is not recommended. Instead, select a base type that already has the `whiteSpace` facet you want. The types `string`, `normalizedString`, and `token` have the `whiteSpace` values `preserve`, `replace`, and `collapse`, respectively. For example, if you wish to define a string-based type that will have its whitespace collapsed, base your type on `token`, instead of basing it on `string` and applying a `whiteSpace` facet. Section 11.2.1 on p. 205 provides a discussion of these three types.

8.5 | Preventing simple type derivation

XML Schema allows you to prevent derivation of other types from your type. By specifying the `final` attribute with a value of `#all` in your simple type definition, you prevent derivation of any kind (restriction, extension, list, or union). If you want more granular control, the value of `final` can be a whitespace-separated list of any of the keywords `restriction`, `extension`, `list`, or `union`. The `extension` value refers to the extension of simple types to derive complex types, described in Section 13.4.1 on p. 306. Example 8–20 shows some valid values for `final`.

Example 8–20. Valid values for the `final` attribute in simple type definitions

```
final="#all"
final="restriction list union"
final="list restriction extension"
final="union"
final=""
```

Example 8–21 shows a simple type that cannot be restricted by any other type or used as the item type of a list. With this definition of `DressSizeType`, it would have been illegal to define `MediumDressSizeType` in Example 8–4 because it attempts to restrict `DressSizeType`.

Example 8–21. Preventing type derivation

```
<xs:simpleType name="DressSizeType" final="restriction list">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="2"/>
    <xs:maxInclusive value="18"/>
  </xs:restriction>
</xs:simpleType>
```

If no `final` attribute is specified, it defaults to the value of the `finalDefault` attribute of the `schema` element. If neither `final` nor `finalDefault` is specified, there are no restrictions on derivation from that type. You can specify the empty string (`""`) for the `final` value if you want to override the `finalDefault` value.

8.6 | Implementation-defined types and facets

Starting with version 1.1, additional simple types and facets may be defined and supported by a particular XML Schema implementation.

8.6.1 *Implementation-defined types*

An implementation can choose to support a set of primitive simple types in addition to those built into XML Schema (described in Chapter 11).

Suppose that an implementation defines a special primitive type `ordinalDate` that represents an ordinal date: a year, followed by a hyphen, followed by a number from 001 to 366 indicating the day of the year. Although an ordinal date value could be represented as a string, it may be beneficial to promote it to its own primitive type if it has special considerations for ordering or validation of its values, or special operations that can be performed on it (for example, subtracting two ordinal dates to get a duration).

A schema author can use an implementation-defined type just like a built-in type, except that it will be in a different namespace defined by the implementation. The schema in Example 8–22

Example 8–22. Using an implementation-defined type

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ext="http://example.org/extensions">
  <xs:element name="anyOrdinalDate" type="ext:ordinalDate"/>
  <xs:element name="recentOrdinalDate" type="OrdinalDateIn2011"/>
  <xs:simpleType name="OrdinalDateIn2011">
    <xs:restriction base="ext:ordinalDate">
      <xs:minInclusive value="2011-001"/>
      <xs:maxInclusive value="2011-365"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

contains two references to the `ordinalDate` type, which is in the hypothetical `http://example.org/extensions` namespace. The `anyOrdinalDate` element declaration refers to the type directly by its qualified name. The `OrdinalDateIn2011` user-defined simple type is a restriction of `ordinalDate` using bounds facets to specify a range of allowed values.

8.6.2 *Implementation-defined facets*

Implementation-defined facets might specify additional constraints on the valid values, or even signal to the processor how to process the value. An example is the Saxon processor's `preprocess` facet which allows you to specify an XPath expression that transforms the value in some way before validation.

In Example 8–23, the `saxon:preprocess` facet appears among the children of `restriction`. You can tell that it is an implementation-defined facet because it is in a different namespace, `http://saxon.sf.net/`. This particular example is telling the processor to convert the value to upper case before validating it against the enumeration facets. It is essentially implementing a case-insensitive enumeration.

Example 8–23. Using the Saxon preprocess facet

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:saxon="http://saxon.sf.net/">
  <xs:simpleType name="SMLXSizeType">
    <xs:restriction base="xs:token">
      <saxon:preprocess action="upper-case($value)"/>
      <xs:enumeration value="SMALL"/>
      <xs:enumeration value="MEDIUM"/>
      <xs:enumeration value="LARGE"/>
      <xs:enumeration value="EXTRA LARGE"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Implementation-defined facets can apply to the XML Schema built-in types (and user-defined restrictions of them); they can also apply to any implementation-defined types such as the `ordinalDate` example type described in the previous section.

While implementation-defined types and facets can be useful, they do affect the portability of your schema. With the schema in Example 8–23, if you try to validate a document that contains lower-case "small" for a size, it would be valid when using Saxon but not when using a different implementation. Therefore, implementation-defined facets should be used only in controlled situations. Section 23.5.3 on p. 642 provides more information on how to make your schemas more portable across implementations when using implementation-defined types and facets.

Index

Symbols

Index entries in gray refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index entries in gray refer to XML Schema 1.1.

K

L

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index entries in gray refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index entries in gray refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.

Index **entries in gray** refer to XML Schema 1.1.