Simple types

FOR PUBLIC RELEASE

Chapter 9

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

9.1 | Simple type varieties

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

1. *Atomic types* have values that are indivisible, such as `10` and `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 11, "Union and list types."

9.1.1 *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 much 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 the instance document.

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 an overkill unless you have a special need to process these items separately.

9.2 | Simple type definitions

9.2.1 *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` or `redefine`) and are required to have a name that is unique among the data types (both simple and complex) in the schema. The XSDL syntax for a named simple type definition is shown in Table 9–1.

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 of the examples of named types in this book have the word "Type" at the end of their names, to clearly distinguish them from element-type names and attribute names. However, this is not a requirement; you may in fact have a data type definition and an element declaration using the same name.

Example 9–1 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.

Table 9–1 XSDL syntax: named simple type definition

Name			
`simpleType`			
Parents			
`schema`, `redefine`			
Attribute name	*Type*	*Required/ default*	*Description*
`id`	`ID`		unique ID
`name`	`NCName`	required	simple type name
`final`	`"#all"` | list of (`"extension"` | `"restriction"` | `"list"` | `"union"`)	defaults to `finalDefault` of `schema`	whether other types can be derived from this one, see Section 9.5
Content			
`annotation?, (restriction | list | union)`			

Example 9–1. Defining and referencing a named simple type

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

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

9.2.2 *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, or redefined. The XSDL syntax to define an anonymous simple type is shown in Table 9–2.

Table 9–2 XSDL syntax: anonymous simple type definition

Name			
`simpleType`			
Parents			
`element`, `attribute`, `restriction`, `list`, `union`			
Attribute name	***Type***	***Required/default***	***Description***
`id`	`ID`		unique ID
Content			
`annotation?, (restriction | list | union)`			

Example 9–2 shows the definition of an anonymous simple type within an element declaration.

Example 9–2. Defining an anonymous simple type

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

9.2.3 *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 `Product-CodeType` 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.

Named types can also make the schema more readable, when the type definitions are complex.

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, 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 try to define a data type named `OneToTenType` that is reused by other unrelated element declarations with the same value space. If the value space for one of the element declarations that uses the named data type changes, but the other element declarations do not change, it actually makes maintenance more difficult, because a new data type needs 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 data type right there with the element or attribute declaration.

9.3 | 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 to add attributes, which results in a complex type. This is described in Section 14.4.1, "Simple content extensions."

Every new simple type restricts the value space of its base type in some way. Example 9–3 shows a definition of `DressSizeType` that restricts the built-in type `integer`.

Example 9–3. Deriving a simple type from a built-in simple type

```
<xsd:simpleType name="DressSizeType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="2"/>
    <xsd:maxInclusive value="18"/>
    <xsd:pattern value="\d{1,2}"/>
  </xsd:restriction>
</xsd: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 9–4.

Example 9–4. Deriving a simple type from a user-derived simple type

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

A simple type restricts its base type by applying facets to restrict its values. In Example 9–4, the facets `minInclusive` and `maxInclu-`

`sive` are used to restrict the value of `MediumDressSizeType` to be between `8` and `12` inclusive.

9.3.1 *Defining a restriction*

The syntax for a `restriction` element is shown in Table 9–3. You must specify one base type either by using the `base` attribute, or by defining the simple type anonymously using a `simpleType` child. The alternative of using a `simpleType` child is generally only useful when restricting list types, as described in Chapter 11, "Union and list types."

Table 9–3 XSDL syntax: simple type restriction

Name			
`restriction`			
Parents			
`simpleType`			
Attribute name	***Type***	***Required/default***	***Description***
`id`	`ID`		unique ID
`base`	`QName`	either a `base` attribute or a `simpleType` child is required	simple type that is being restricted
Content			
`annotation? , simpleType? , (minExclusive | minInclusive | maxExclusive | maxInclusive | length | minLength | maxLength | totalDigits | fractionDigits | enumeration | pattern | whiteSpace)*`			

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` and `enumeration`. 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.

9.3.2 *Overview of the facets*

The available facets are listed in Table 9–4.

The XSDL syntax for applying a facet is shown in Table 9–5. All facets must have a `value` attribute, which has different valid values

Table 9–4 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

depending on the facet. Most facets may also have a fixed attribute, as described in Section 9.3.4, "Fixed facets."

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 string, the length facet is also applicable to your new type. Section 9.4, "Facets," describes each of the facets in detail and lists the built-in types to which the facet can apply.

9.3.3 *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 9–4 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. Because MediumDressSizeType inherits all of the facets from DressSizeType, this same pattern facet applies to MediumDressSizeType also. Example 9–5 shows an equivalent definition of MediumDressSizeType, where it restricts integer and has the pattern facet applied.

Sometimes a simple type definition will include facets that are also specified for one of its ancestors. In Example 9–4, MediumDressSizeType includes minInclusive and maxInclusive, which are also applied to its base type, DressSizeType. The minInclusive and maxInclusive facets of MediumDressSizeType (whose values are

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.

Table 9–5 XSDL syntax: facet

Name			
minExclusive, minInclusive, maxExclusive, maxInclusive, length, minLength, maxLength, totalDigits, fractionDigits, enumeration, pattern, whiteSpace			
Parents			
restriction			
Attribute name	*Type*	*Required/default*	*Description*
id	ID		unique ID
value	various	required	value of the restricting facet
fixed	boolean	false; n/a for pattern, enumeration	whether the facet is fixed and therefore cannot be restricted further, see Section 9.3.4
Content			
annotation?			

Example 9–5. Effective definition of MediumDressSizeType

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

8 and 12, respectively) override those of DressSizeType (2 and 18, respectively).

It is a requirement that the facets of the derived type (in this case MediumDressSizeType) be more restrictive than those of the base type. In Example 9–6, 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 9–6. Illegal attempt to extend a simple type

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

This rule also applies when you are restricting the built-in types. For example, the `short` data 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 9.4.4, “Enumeration.”

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

9.3.4 *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` and `enumeration`. Example 9–7 shows our `DressSize-`

`Type` with fixed `minExclusive` and `maxInclusive` facets, as indicated by a `fixed` attribute that is set to `true`.

Example 9–7. Fixed facets

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

With this definition of `DressSizeType`, it would have been illegal to define the `MediumDressSizeType` as shown in Example 9–4 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`.

9.3.4.1 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, such as, for example, a `SalePrice` type whose values must end in `99`. However, they cannot modify the `fractionDigits` of the type, because this would result in a type not representing a price with both dollars and cents.

9.4 | Facets

9.4.1 *Bounds facets*

The four bounds facets (`minInclusive`, `maxInclusive`, `minExclusive`, and `maxExclusive`) restrict a value to a specified range. Our previous examples apply `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 values that are outside 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 `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 types derived from them. Special consideration should be given to time zones when applying bounds facets to date and time types. For more information, see Section 12.4.12, "Date and time ordering."

9.4.2 *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 legacy 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 in number of items in the list. The facet value for `length` must be a non-negative 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 non-negative integers.

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

9.4.2.1 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`, and `base64-Binary` do not allow an empty value, unless `xsi:nil` appears in the element tag.

There may be a case where you 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 data type that includes both the integer type and an empty string. For example:

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

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

The `length` facet only applies to the string-based types, the legacy types, the binary types, and `anyURI`. It does not make sense to try to limit the length of the date and 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 9.4.1, "Bounds facets." You can also control the number of significant digits in an integer using the `totalDigits` facet, as discussed in Section 9.4.3, "`totalDigits` and `fractionDigits`." However, these facets do not consider leading zeros to be significant. Therefore, they cannot force the integer to appear in the instance as 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 12.3.3.1, "Design hint: Is it an integer or a string?" for a discussion of this issue.

9.4.3 `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 non-negative 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, and types derived from them. The `fractionDigits` facet may only be applied to `decimal`, because it is fixed at 0 for all integer types.

9.4.4 *Enumeration*

The `enumeration` facet allows you to specify a distinct set of valid values for a type. Unlike most other facets (except `pattern`), 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 data 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 9–8 shows a simple type `SMLXSizeType` that allows the values `small`, `medium`, `large`, and `extra large`.

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 9–9.

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 9–10 attempts to add an `enumeration` facet to allow the value `extra small`. This type definition

Example 9–8. Applying the enumeration facet

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

Example 9–9. Restricting an enumeration

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

is illegal because it attempts to extend rather than restrict the value space of `SMLXSizeType`.

Example 9–10. Illegal attempt to extend an enumeration

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

The only way to add an enumeration value to a type is by defining a union type. Example 9–11 shows a union type that adds the value `extra small` to the set of valid values. Union types are described in detail in Chapter 11, "Union and list types."

Example 9–11. Using a union to extend an enumeration

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

When enumerating numbers, it is important to note that the `enumeration` facet works on the actual value of the number, not its lexical representation as it appears in an XML instance. Example 9–12 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 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 12.7, "Type equality."

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

9.4.5 *Pattern*

The `pattern` facet allows you to restrict values to a particular pattern, represented by a regular expression. Chapter 10, "Regular expressions," provides more detail on the rules for the regular expression syntax. Unlike most other facets (except `enumeration`), 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 9–12. Enumerating numeric values

Schema:

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

Valid instances:

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

Example 9–13 shows a simple type `DressSizeType` that includes the pattern `\d{1,2}`, which restricts the size to one or two digits.

Example 9–13. Applying the pattern facet

```
<xsd:simpleType name="DressSizeType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="2"/>
    <xsd:maxInclusive value="18"/>
    <xsd:pattern value="\d{1,2}"/>
  </xsd:restriction>
</xsd: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 9–14, 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 9–14. Restricting a pattern

```
<xsd:simpleType name="SmallDressSizeType">
  <xsd:restriction base="DressSizeType">
    <xsd:minInclusive value="2"/>
    <xsd:maxInclusive value="6"/>
    <xsd:pattern value="\d{1}"/>
  </xsd:restriction>
</xsd: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 facet of both the type and its ancestors, ensuring that it is in fact a subset. Example 9–15 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`.

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.

Example 9–15. Illegal attempt to extend a pattern

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

9.4.6 *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 as to what to do with whitespace. The valid values for the `whiteSpace` facet are:

- `preserve`: All whitespace is preserved; the value is not changed. This is how XML 1.0 processors handle whitespace in the character data content of elements.
- `replace`: Each occurrence of a tab (`#x9`), line feed (`#xA`), and carriage return (`#xD`) is replaced with a single space (`#x20`). This is how XML 1.0 processors handle whitespace in attributes of type `CDATA`.
- `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. This is how XML 1.0 processors handle whitespace in all attributes that are not of type `CDATA`.

Table 9–6 shows examples of how values of a string-based type will be handled depending on its `whiteSpace` facet.

Table 9–6 **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 9–8, the base type of SMLXSizeType is token, which has a whiteSpace facet of collapse. Example 9–16 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 9–16. 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 data 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 your type on string and applying a whiteSpace facet. Section 12.2.1, "string, normalizedString, and token," provides a discussion of these three types.

9.5 | Preventing simple type derivation

XML Schema allows you to prevent derivation of other types from your type. By specifying the final attribute in your simple type definition, you may prevent derivation of any kind (restriction, extension, list, or union) by specifying a value of `#all`. If you want more specific control, the value of `final` can be a whitespace-separated list of any of the keywords `restriction`, `extension`, `list`, or `union`. Extension refers to the extension of simple types to derive complex types, described in Chapter 14, "Deriving complex types." Example 9–17 shows some valid values for `final`.

Example 9–17. Valid values for the `final` attribute in simple type definitions

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

Example 9–18 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 9–4 because it attempts to restrict `DressSizeType`.

Example 9–18. Preventing type derivation

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