Document Type Definitions (DTDs)

Although you can have your own custom XML element tags, in order to make everything really useful you will want to have a way to define the structure, valid elements, values, and attributes within your XML. This ensures that, although your tag names are your own, you can still parse the data easily and in a way that makes sense (and ensure that all nodes are valid nodes which are spelled correctly). In order to grab the correct data, you need to know what is nested within a particular element.

For example, if we look at the following portion of XML:

```
<name>
  <first>John</first>
  <last>Doe</last>
</name>
```

This structure makes sense because a name is made up of a first name and last name. It is still well-formed XML to put:

```
<first>
  <name>John</name>
  <last>Doe</last>
</first>
```

But the above is nonsensical, so we need a way to ensure that data is structured (nested) properly and valid. There are a few ways to enforce element names and structure:

- DTD (**D**ocument **T**ype **D**efinition)
- Schemas

We'll be focusing on DTDs in this document. A DTD is a document (***.dtd*** is the file extension) which contains element type declarations, attribute list declarations, entity declarations and notation declarations. It essentially *defines* your custom element names and any expected attributes or value types. You specify the definition with a DOCTYPE declaration in your XML file. (Note that DTDs are an older technology but it has widespread usage so you need to know it.)

In HTML, that's what <!DOCTYPE html> means. It essentially is telling a parser that the document is of type "html" and should be validated using the "html" spec. This is how parsers determine which elements are valid and how elements should be nested. There is a DTD defining the usage/nesting of HTML elements. Since the HTML spec is a standard, browsers know where to go to find the HTML DTD. As you are creating your own DTD, the <!DOCTYPE> line will look a little different.

Note

Although HTML5 uses a <!DOCTYPE> the actual validation is via a Schema because XML Schemas are more powerful and flexible than a DTD. <!DOCTYPE> is still used in the HTML so that there's not too much change to the way an HTML file is structured. There is an actual HTML4 DTD file. (Try looking it up.)

For inline DTDs, you will need to add the following inside your XML file, **<u>after the XML declaration but *before* the XML code</u>**:

```
<!DOCTYPE root[
  ...
]>
```

where the *root* is the name of your XML file's root element.

Element definitions

To define an element, use the following syntax:

```
<!ELEMENT elementname ([elementcontent])>
```

This defines an element called "elementname". Within the parentheses, you need to define what type of content this element will hold.

Element content

Elements containing only <u>other elements</u> have <u>*element content*</u>. The eligible element content names should be listed within the parentheses. This would be expressed as:

```
<!ELEMENT elementname (childelementname)>
```

You can define multiple elements as content. For example:

```
<!ELEMENT elementname (childelement1, childelement2)>
```

In the above line, elements with the names "childelement1" and "childelement2" are listed with no wildcards meaning they <u>each must appear and should appear only once</u>. Separating elements with commas also defines a **<u>sequence</u>**, so elements *must* appear in the order given. This means that the structure is:

```
<elementname>
  <childelement1></childelement1>
  <childelement2></childelement2>
</elementname>
```

There are special characters to specify if elements are to appear more than one or if elements are optional. These special characters are placed after the element they apply to. See the table below for the special characters (note their usage and similarities to their usage in regular expressions).

Wildcard character	***Meaning***
*	The element may appear **<u>zero or more times</u>**.
+	The element will appear **<u>one or more times</u>**. Note that the element <u>must</u> appear at least once.

?	The element is optional and may appear **<u>once or not at all</u>**. (If it may appear more than once, use the asterisk.)

So if you want the `<childelement1>` element to appear one or more times and have the `<childelement2>` element appear only once and always once, the `<elementname>` declaration now becomes:

```
<!ELEMENT elementname (childelement1+, childelement2)>
```

Parseable character data

Elements containing data (numbers and text, for example) and not other elements contain *<u>parseable character data</u>*. To declare this type of content, we use the keyword `#PCDATA` (**<u>P</u>**arseable **<u>C</u>**haracter **<u>DATA</u>**).

For example, if you had an element called "fullname" and it contained text, you would declare this element with:

```
<!ELEMENT fullname (#PCDATA)>
```

<u>Other possible content</u>

It is possible to declare empty elements. To do this, use the `EMPTY` keyword like so:

```
<!ELEMENT elementname EMPTY>
```

Empty elements are self-closing tags (recall the `<img>` element in HTML).

Mixed content

Elements containing both character data *and* other elements have *<u>mixed content</u>*. An example of this, is a nested list in HTML. See below:

```
<ul>
  <li>List item 1</li>
  <li>List item 2
    <ul>
      <li>Sublist item 1</li>
      <li>Sublist item 2</li>
      <li>Sublist item 3</li>
    </ul>
  </li>
</ul>
```

Recall that in nested lists, the sublist is opened <u>within its parent list item</u> (i.e. before the parent list item is closed). This means that <li> elements can contain both character data (#PCDATA) and other elements (<ol> or <ul>). Of course the sublist element will only appear once if it does.

To declare and element with mixed content (using the above HTML example):

```
<!ELEMENT li (#PCDATA | ol | ul)*>
```

Notice there is a special character: the pipe character ("|"). This indicates an OR relationship.

Important!

Mixed content must **always separate the #PCDATA and possible elements with an OR ("|") and the entire group must always be optional ("*")**. The **#PCDATA must always come before element names in the grouping**.

Example. Take a look at the previous example shown again below (***people.xml***).

```
<?xml version="1.0" encoding="utf-8" ?>
<people>
  <person>
    <name>
      <first>John</first>
      <last>Doe</last>
    </name>
  </person>
  <person>
    <name>
      <first>Harry</first>
      <last>Potter</last>
    </name>
  </person>
</people>
```

You can put your declarations in the same XML file or you can put your declarations in a separate DTD file (this is recommended because it looks cleaner).

To put your declarations in the same XML file:

1. Open ***people.xml*** in your plain-text editor of choice if you don't already have it open.
2. Above your XML code but below the XML declaration, place the following:

   ```
   <!DOCTYPE people[

   ]>
   ```

   This opens and declares a DOCTYPE called `people` (this **must be your root element name**). The definition of the *people* doctype will be placed within the square brackets [...].

3. The <people> element only contains <person> elements as its children, so it only has element content. There can be multiple <person> elements within <people>. Within the doctype square brackets, add:

   ```
   <!ELEMENT people (person+)>
   ```

4. Within each <person> element there is only one <name> element. Add the following on a new line:

   ```
   <!ELEMENT person (name)>
   ```

5. Within each <name> element, there are only <first> and <last> elements, each always appearing only once. Add:

   ```
   <!ELEMENT name (first, last)>
   ```

6. The <first> and <last> elements contain character data, so add:

   ```
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last (#PCDATA)>
   ```

7. You should now have the following as your DOCTYPE declaration:

   ```
   <!DOCTYPE people[
     <!ELEMENT people (person+)>
     <!ELEMENT person (name)>
     <!ELEMENT name (first, last)>
     <!ELEMENT first (#PCDATA)>
     <!ELEMENT last (#PCDATA)>
   ]>
   ```

Creating a DTD file:

The "standalone" attribute

A caution about the **standalone** attribute as you may come across it when you read about XML:

When you use your own DTD, you can indicate in the XML file itself that a custom DTD (your inline DTD or DTD file) will be used for validation **ONLY**. **This is only for your knowledge, but is not necessary in order to use your custom DTD for validation!** Using the following **standalone** attribute will cause errors if you don't take into account whitespace (e.g. indents for formatting). Without the attribute it will still work, so **you can safely ignore this for your purposes**. The **standalone** attribute is only used for special cases where you have defined an extra-specific structure and you take into account whitespace and exact formatting. Using **standalone="yes"** means that standard XML validation (i.e. XML standard spec) will not be used in conjunction with your DTD, which you do **NOT** want in most cases.

To do this, you need to modify the first line of the XML file from

```
<?xml version="1.0" encoding="utf-8" ?>
```

to

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
```

(The new addition to the line has been bolded.) The `standalone` attribute indicates that an external DTD is to be used. If you are **not** using a custom DTD, you DO NOT need to include the standalone attribute. By default, the value for standalone is "no" so you **only need to include the `standalone` attribute if the value is "yes"**.

For our purposes, you do NOT need to use the standalone attribute at all.

1. Open your plain-text editor of choice and create a new file, saving it as ***people.dtd***.
2. The <people> element only contains <person> elements as its children, so it only has element content. There can be multiple <person> elements within <people>. Add the following in your DTD file (you don't need the "<!DOCTYPE [" and "]>" bits inside of an external DTD file):

   ```
   <!ELEMENT people (person+)>
   ```

3. Within each <person> element there is only one <name> element. Add the following after the above (on a new line):

   ```
   <!ELEMENT person (name)>
   ```

4. Within each <name> element, there are only <first> and <last> elements, each always appearing only once. Add to the DTD file (on a new line):

   ```
   <!ELEMENT name (first, last)>
   ```

5. Finally, the <first> and <last> elements contain character data. Add the following to the file (again on their own lines):

   ```
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last (#PCDATA)>
   ```

6. Save the file. If you have not yet named the file, name it as ***people.dtd***. At this point, your DTD file should contain:

   ```
   <!ELEMENT people (person+)>
   <!ELEMENT person (name)>
   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last (#PCDATA)>
   ```

7. Now you need to refer to this DTD file in your XML file. Above your XML code but directly below the XML declaration, add the following line to ***people.xml***:

   ```
   <!DOCTYPE people SYSTEM "people.dtd">
   ```

   The above line declares the XML file to be of the document type *people* and states that the definition is found in a local file (`SYSTEM`) at ***people.dtd***. The "`people.dtd`" part is the path to the DTD file, so "`people.dtd`" implies that the DTD file is in the same directory as the current file (***people.xml***).

Attribute definitions

Now that you've learned how to create a DTD and define element types, you need to learn how to define valid attributes for your custom elements within your DTD. To do this, you need to declare an **attribute list** (`ATTLIST`) for an element with attributes.

The syntax to declare an attribute list for an element is:

```
<!ATTLIST [element_name] [attribute_name] [attr_type] [value_or_required]>
```

There are four main required pieces of information:

- the element name/type for which the attribute applies
- the name of your custom attribute
- the type of data the attribute value is (e.g. if the attribute must be unique, this is an `ID` type)
- if there is a value, define the default; if there's no given value specify whether or not the attribute is required (e.g. in the HTML spec, <img> elements require the **alt** attribute)

Element name

This is simply the element name the attribute should be associated with. For example, if you wanted the `<name>` element to have an attribute called "nickname" you would put "name" as the element name and "nickname" as the attribute name.

Attribute name

This is what your attribute will be called.

Attribute value type

This is the *type of data* the attribute value will hold. See the possible attribute types in the table below.

Type	***Notes***
`CDATA`	`CDATA` is used for strings, so any text is allowed (whether text or numbers). **Note**: Do **NOT** confuse this with an XML CDATA section. A CDATA attribute type for DTDs and a CDATA section in an XML document are different things. *Example*: If you have an attribute called **nickname** which would hold a person's nickname, this is text data so it would be of `CDATA` type.
`ID`	This is used for values which should be **unique within the**

	document. Note that an ID type still follows naming rules (i.e. an ID type still requires that the id value starts with a letter and cannot start with a number). *Example*: If you have an attribute called **productid** which you want to hold unique values (IDs should be unique, by definition), use the `ID` type.
`IDREF` (or **`IDREFS`** if multiple)	This is used for values which refer to **another ID within the document**. When the attribute is used, multiple IDs are separated by spaces. Recall, in HTML, the **for** attribute for a `<label>` element refers to an ID of a form control. This is an `IDREF` type of attribute.
`ENTITY` (or **`ENTITIES`** if multiple)	This refers to data or an external location. When referring to data, you use this to define your own entity references (think `&amp;`). Note that when you use this to define your own entity, you need to use this in conjunction with an entity declaration (see page 9). *Example*: For an `<img>` element, there is a **src** attribute which refers to some file (or, entity) somewhere. `<!ATTLIST img src ENTITY #REQUIRED>`
`NMTOKEN` (or **`NMTOKENS`** if multiple)	Name tokens are string tokens. Multiple tokens (`NMTOKENS`) are separated by spaces.

Value or required

You can specify whether attributes are required or optional and also specify a default value.

There are four possible values:

Value	***Notes***
`#REQUIRED`	This means that the attribute must exist. If the attribute is not specified, an error will be thrown upon validation. There is no specified value but the attribute must be there. *Example*: Think of the **alt** attribute for images. This is a required attribute because you will get a validation error if it does not exist. `<!ATTLIST img alt CDATA #REQUIRED>`
`#IMPLIED`	This means that the attribute is **optional**. There is no specified value.

	<u>Example</u>: An <h1> element can have a **class** attribute, but it's optional. `<!ATTLIST h1 class CDATA #IMPLIED>`
"*value*"	This is just a value, not a keyword. This is used as the default value for an attribute. This is usually used in conjunction with an enumerated list of possible values. *<u>Example</u>*: An <input> element by default is a text field if you do not specify the type. `<!ATTLIST input type (text | submit | reset | checkbox | radio) "text">`
`#FIXED` "*value*"	This means that the attribute can only have a value of "*value*" (whatever you put for the value). The attribute is not required, but if you use it, it must match the given value. *<u>Example</u>*: An <option> element has a **selected** attribute for selected options. The attribute is optional but if it's used, the only allowable value is "selected" (i.e. **selected="selected"**). `<!ATTLIST option selected CDATA #FIXED "selected">`

<u>Note</u>

All attribute values are *normalized* by XML processors. This means that character references are replaced by their reference values, entities are resolved, and extra whitespace is stripped out.

Declaring multiple attributes for the same element type

When you have multiple attributes for the same element type, you do not add multiple `<!ATTLIST ... >` declarations. Instead, put **<u>one</u>** declaration like so:

```
<!ATTLIST elementname
  attr1 ID #REQUIRED
  attr2 CDATA #IMPLIED
  attr3 CDATA #REQUIRED
  ...>
```

<u>Important</u>

Although parsers don't treat namespace declarations as attributes, <u>within your DTDs they *are* like attributes</u>, so declare them if you're using namespaces. This is because DTD syntax was completed before namespaces. Namespaces don't always play nice with DTDs. Namespaces work much better with Schemas.

Entity definitions

Recall that you can have attributes of type ENTITY, which you can define as your own entity reference. For this type of entity reference, you need to define the *data or content* that will be referenced when the entity reference is used. For this you need an entity declaration. An entity declaration takes the form of:

`<!ENTITY entityname` *[entitydata]*>

You can then use the entity reference in the XML code with `&entityname;`. When the XML is processed by a parser, the entity reference is replaced by the data defined in your entity declaration.

Take a look at the code excerpt below:

(in the DTD part)
```
<!ENTITY hello "Hello? Is it me you're looking for?">
```

(in the XML)
```
<song>&hello;</song>
```

When the XML is processed by the parser, `&hello;` will be replaced by "`Hello? Is it me you're looking for?`"

Validation

To validate your XML against your custom DTD you *can* come up with a code solution, but for now we'll look at using an online validator to quickly check your XML.

Validate your XML

Beyond the primitive method of opening your XML file in a browser and seeing if anything breaks, you can use the XML validator (https://www.xmlvalidation.com) to check to see if your XML is well-formed.

Validate your XML against your custom DTD

If you want to do a quick check, you can use the online validator to validate your XML. The online validator (above) can validate using both internal and external DTDs.

Validate using a plugin in your code editor

VSCode: Install and use the XML plugin for VSCode by Redhat. This plugin supports both DTD and Schema validation.

Notepad++: If you don't have the XML Tools plugin for Notepad++, you can install it. Just search for the **XML Tools** plugin for Notepad++. Make sure you download the correct version for your Notepad++ installation (32-bit or 64-bit). Make sure you follow the instructions in the installation readme to ensure you have installed all dependencies correctly.

Once you have the XML Tools plugin installed, go to **Plugins > XML Tools > Validate now** to validate against the DTD.

If you do not use Notepad++, take a look at which plugins are available for your text/code editor of choice. There may be an XML plugin available.