

XPointers

XPointer, the XML Pointer Language, defines an addressing scheme for individual parts of an XML document. XLinks point to a URI (in practice, a URL) that specifies a particular resource. The URI may include an XPointer part that more specifically identifies the desired part or element of the targeted resource or document. This chapter discusses XPointers.



Caution

This chapter is based on the March 3, 1998 working draft of the XPointer specification. The broad picture presented here is likely to be correct but the details are subject to change. You can find the latest working draft at <http://www.w3.org/TR/WD-xptr>.

Why Use XPointers?

URLs are simple and easy to use, but they're also quite limited. For one thing, a URL only points at a single, complete document. More granularity than that, such as linking to the third sentence of the 17th paragraph in a document, requires the author of the targeted document to manually insert named anchors at the targeted location. The author of the document doing the linking can't do this unless he or she also has write access to the document being linked to. Even if the author doing the linking can insert named anchors into the targeted document, it's almost always inconvenient.

It would be more useful to be able to link to a particular element or group of elements on a page without having to change the document you're linking to. For example, given a large page such as the complete baseball statistics of Chapters 4 and 5, you might want to link to only one team or one player. There are several parts to this problem. The first part is addressing the individual elements. This is the part that XPointers solve. XPointers allow you to target a given element by number, name, type, or relation to other elements in the document.



In This Chapter

Why use XPointers?

XPointer examples

Absolute location terms

Relative location terms

Relative location term arguments

String location terms

The origin absolute location term

Purpose of spans



The second part of the problem is the protocol by which a browser asks a Web server to send only part of a document rather than the whole thing. This is an area of active research and speculation. More work is needed. XPointers do little to solve this problem, except for providing a foundation on which such systems can build. For instance, the best effort to date are the so-called “byte range extensions to HTTP” available in HTTP 1.1. So far these have not achieved widespread adoption, mostly because Web authors aren’t comfortable specifying a byte range in a document. Furthermore, byte ranges are extremely fragile. Trivial edits to a document, even simple reformatting, can destroy byte range links. HTTP 1.1 does allow other range units besides raw bytes (for example, XML elements), but does not require Web servers or browsers to support such units. Much work remains to be done.

The third part of the problem is making sure that the retrieved document makes sense without the rest of the document to go along with it. In the context of XML, this effectively means the linked part is well-formed or perhaps valid. This is a tricky proposition, because most XML documents, especially ones with nontrivial prologs, don’t decompose well. Again, XPointers don’t address this. The W3C XML Fragment Working Group is addressing this issue, but the work is only just beginning.

For the moment, therefore, an XPointer can be used as an index into a complete document, the whole of which is loaded and then positioned at the location identified by the XPointer. In the long-term, extensions to both XML, XLink, HTTP, and other protocols may allow more sophisticated uses of XPointers. For instance, you might be able to quote a remote document by including only an XLink with an XPointer to the paragraph you want to quote, rather than retyping the text of the quote. You could include cross-references inside a document that automatically update themselves as the document is revised. These uses, however, will have to wait for the development of several next-generation technologies. For now, we must be content with precisely identifying the part of a document we want to jump to when following an XLink.

XPointer Examples

HTML links generally point to a particular document. Additional granularity, that is, pointing to a particular section, chapter, or paragraph of a particular document, isn’t well-supported. Provided you control both the linking and the linked document, you can insert a named anchor into an HTML file at the position to which you want to link. For example:

```
<H2><A NAME="xpointers">XPointers</A></H2>
```

You can then link to this particular position in the file by adding a # and the name of the anchor into the link. For example, in a table of contents you might see:

```
<A HREF="#xpointers">XPointers</A>
```

In practice, this solution is kludgy. It's not always possible to modify the target document so the source can link to it. The target document may be on a different server controlled by someone other than the author of the source document. And the author of the target document may change or move it without notifying the author of the source.

Furthermore, named anchors violate the separation of markup from content. Placing a named anchor in a document says nothing about the document or its content. It's just a marker for other documents to refer to. It adds nothing to the document's own content.

XLinks allow much more sophisticated connections between documents through the use of XPointers. An XPointer can refer to a particular element of a document; to the first, second, or 17th such element; to the first element that's a child of a given element; and so on. XPointers provide extremely powerful connections between documents. They do not require the targeted document to contain additional markup just so its individual pieces can be linked to.

Furthermore, unlike HTML anchors, XPointers don't point to just a single point in a document. They can point to ranges or spans. Thus, you can use an XPointer to select a particular part of a document, perhaps so it can be copied or loaded into a program.

Here are a few examples of XPointers:

```
root()
id(dt-xmldecl)
descendant(2,termref)
following(,termdef,term,CDATA Section)
html(recent)
id(NT-extSubsetDecl)
```

Each of these selects a particular element in a document. The document is not specified in the XPointer; rather, the XLink specifies the document. The XLinks you saw in the previous chapter did not contain XPointers, but it isn't hard to add XPointers to them. Most of the time you simply append the XPointer to the URI separated by a #, just as you do with named anchors in HTML. For example, the above list of XPointers could be suffixed to URLs and come out looking like the following:

```
http://www.w3.org/TR/1998/REC-xml-19980210.xml#root()
http://www.w3.org/TR/1998/REC-xml-19980210.xml#id(dt-xmldecl)
http://www.w3.org/TR/1998/REC-xml-19980210.xml#descendant(2,termref)
http://www.w3.org/TR/1998/REC-xml-19980210.xml#following(,termdef,term,CDATA Section)
http://www.w3.org/TR/1998/REC-xml-19980210.xml#id(NT-extSubsetDecl)
```

Normally these are used as values of the `href` attribute of a `locator` element. For example:

```
<locator
  href="http://www.w3.org/TR/1998/REC-xml-19980210.xml#root()">
  Extensible Markup Language (XML) 1.0
</locator>
```

You can use a vertical bar (|) instead of a # to indicate that you do not want the entire document. Instead, you want only the part of the document referenced by the XPointer. For example:

```
http://www.w3.org/TR/1998/REC-xml-19980210.xml|root()
http://www.w3.org/TR/1998/REC-xml-19980210.xml|id(dt-xmldecl)
http://www.w3.org/TR/1998/REC-xml-19980210.xml|descendant(2,termref)
http://www.w3.org/TR/1998/REC-xml-19980210.xml|following(,termdef,term,CDATA Section)
http://www.w3.org/TR/1998/REC-xml-19980210.xml|id(NT-extSubsetDecl)
```

Whether the client is able to retrieve only a piece of the document is protocol dependent. Most current Web browsers and servers aren't able to handle the sophisticated requests that these XPointers imply. However, this can be useful for custom protocols that use XML as an underlying transport mechanism.

Absolute Location Terms

XPointers are built from *location terms*. Each location term specifies a point in the targeted document, generally relative to some other well-known point such as the start of the document or another location term. The type of location term is given by a keyword such as `id()`, `root()`, or `child()`.

Some location terms take arguments between the parentheses. To demonstrate the point, it's useful to have a concrete example in mind. Listing 17-1 is a simple, valid document that should be self-explanatory. It contains information about two related families and their members. The root element is `FAMILYTREE`. A `FAMILYTREE` can contain `PERSON` and `FAMILY` elements. Each `PERSON` and `FAMILY` element has a required `ID` attribute. Persons contain a name, birth date, and death date. Families contain a husband, a wife, and zero or more children. The individual persons are referred to from the family by reference to their IDs. Any child element may be omitted from any element.



Cross-Reference

This XML application is revisited in Chapter 23, *Designing a New XML Application*.

Listing 17-1: A family tree

```

<?xml version="1.0"?>
<!DOCTYPE FAMILYTREE [

  <!ELEMENT FAMILYTREE (PERSON | FAMILY)*>

  <!-- PERSON elements -->
  <!ELEMENT PERSON (NAME*, BORN*, DIED*, SPOUSE*)>
  <!ATTLIST PERSON
    ID      ID      #REQUIRED
    FATHER  CDATA  #IMPLIED
    MOTHER  CDATA  #IMPLIED
  >
  <!ELEMENT NAME (#PCDATA)>
  <!ELEMENT BORN (#PCDATA)>
  <!ELEMENT DIED (#PCDATA)>
  <!ELEMENT SPOUSE EMPTY>
  <!ATTLIST SPOUSE IDREF IDREF #REQUIRED>

  <!--FAMILY-->
  <!ELEMENT FAMILY (HUSBAND?, WIFE?, CHILD*) >
  <!ATTLIST FAMILY ID ID #REQUIRED>

  <!ELEMENT HUSBAND EMPTY>
  <!ATTLIST HUSBAND IDREF IDREF #REQUIRED>
  <!ELEMENT WIFE EMPTY>
  <!ATTLIST WIFE IDREF IDREF #REQUIRED>
  <!ELEMENT CHILD EMPTY>
  <!ATTLIST CHILD IDREF IDREF #REQUIRED>

]>
<FAMILYTREE>

  <PERSON ID="p1">
    <NAME>Domeniquette Celeste Baudean</NAME>
    <BORN>11 Feb 1858</BORN>
    <DIED>12 Apr 1898</DIED>
    <SPOUSE IDREF="p2"/>
  </PERSON>

  <PERSON ID="p2">
    <NAME>Jean Francois Bellau</NAME>
    <SPOUSE IDREF="p1"/>
  </PERSON>

  <PERSON ID="p3" FATHER="p2" MOTHER="p1">
    <NAME>Elodie Bellau</NAME>
    <BORN>11 Feb 1858</BORN>

```

Continued

Listing 17-1 (continued)

```

    <DIED>12 Apr 1898</DIED>
    <SPOUSE IDREF="p4"/>
  </PERSON>

  <PERSON ID="p4" FATHER="p2" MOTHER="p1">
    <NAME>John P. Muller</NAME>
    <SPOUSE IDREF="p3"/>
  </PERSON>

  <PERSON ID="p7">
    <NAME>Adolf Eno</NAME>
    <SPOUSE IDREF="p6"/>
  </PERSON>

  <PERSON ID="p6" FATHER="p2" MOTHER="p1">
    <NAME>Maria Bellau</NAME>
    <SPOUSE IDREF="p7"/>
  </PERSON>

  <PERSON ID="p5" FATHER="p2" MOTHER="p1">
    <NAME>Eugene Bellau</NAME>
  </PERSON>

  <PERSON ID="p8" FATHER="p2" MOTHER="p1">
    <NAME>Louise Pauline Bellau</NAME>
    <BORN>29 Oct 1868</BORN>
    <DIED>11 May 1879</DIED>
    <SPOUSE IDREF="p9"/>
  </PERSON>

  <PERSON ID="p9">
    <NAME>Charles Walter Harold</NAME>
    <BORN>about 1861</BORN>
    <DIED>about 1938</DIED>
    <SPOUSE IDREF="p8"/>
  </PERSON>

  <PERSON ID="p10" FATHER="p2" MOTHER="p1">
    <NAME>Victor Joseph Bellau</NAME>
    <SPOUSE IDREF="p11"/>
  </PERSON>

  <PERSON ID="p11">
    <NAME>Ellen Gilmore</NAME>
    <SPOUSE IDREF="p10"/>
  </PERSON>

  <PERSON ID="p12" FATHER="p2" MOTHER="p1">
    <NAME>Honore Bellau</NAME>
  </PERSON>

```

```

<FAMILY ID="f1">
  <HUSBAND IDREF="p2"/>
  <WIFE IDREF="p1"/>
  <CHILD IDREF="p3"/>
  <CHILD IDREF="p5"/>
  <CHILD IDREF="p6"/>
  <CHILD IDREF="p8"/>
  <CHILD IDREF="p10"/>
  <CHILD IDREF="p12"/>
</FAMILY>

<FAMILY ID="f2">
  <HUSBAND IDREF="p7"/>
  <WIFE IDREF="p6"/>
</FAMILY>

</FAMILYTREE>

```

In sections that follow, this document is assumed to be present at the URL <http://www.theharolds.com/genealogy.xml>. This isn't a real URL, but the emphasis here is on selecting individual parts of a document rather than a document as a whole.

id()

The `id()` location term is one of the simplest and most useful location terms. It selects the element in the document that has an `ID` type attribute with a specified value. For example, consider the URI [http://www.theharolds.com/genealogy.xml#id\(p12\)](http://www.theharolds.com/genealogy.xml#id(p12)). If you look back at Listing 17-1, you find this element:

```

<PERSON ID="p12" FATHER="p2" MOTHER="p1">
  <NAME>Honore Bellau</NAME>
</PERSON>

```

Because `ID` type attributes are unique, you know there aren't other elements that match this XPointer. Therefore, [http://www.theharolds.com/genealogy.xml#id\(p12\)](http://www.theharolds.com/genealogy.xml#id(p12)) must refer to Honore Bellau's `PERSON` element. Note that the XPointer selects the entire element to which it refers, including all its children, not just the start tag.

The disadvantage of the `id()` location term is that it requires assistance from the targeted document. If the element you want to point to does not have an `ID` type attribute, you're out of luck. If other elements in the document have `ID` type attributes, you may be able to point to one of them and use a relative XPointer (discussed in the next section) to point to the one you really want. Nonetheless, `ID` type attributes are best when you control both the targeted document and the linking document, so you can ensure that the IDs match the links even as the documents evolve and change over time.

In some cases, such as a document without a DTD, a targeted document may not have any `ID` type attributes, although it may have attributes named `ID`. In this case, the application may (or may not) try to guess which element you were pointing at. Generally it selects the first element in the document with an attribute of any type and a name whose value matches the requested `ID`. On the other hand, the application is free not to select any element.

root()

The `root()` location term points to the root element of the document. It takes no arguments. For example, the root element of the XML 1.0 specification at <http://www.w3.org/TR/REC-xml> is `spec`. Thus, to select it you can use this URI:

```
http://www.w3.org/TR/REC-xml#root()
```

The `root()` location term is primarily useful in compound XPointers as a basis from which to start. In fact, if no absolute location term is included in a compound location term, `root()` is assumed. However, `root()` can also be used to select the entire document in a URI that uses `|` to indicate that only a part is normally loaded. For example:

```
http://www.w3.org/TR/1999/REC-xml-names-19990114/xml-names.xml|root()
```

html()

The `html()` location term selects named anchors in HTML documents. It has a single argument, the name of the anchor to which it refers. For example, the following named anchor exists in the file <http://metalab.unc.edu/xml/>:

```
<a name="quote"><font color="#AA0000">Quote of the Day</font></a>
```

The XPointer that refers to this element is:

```
http://metalab.unc.edu/xml#html(quote)
```

The `html()` location term primarily exists for backwards compatibility, that is, to allow XLinks to refer to HTML documents. Named anchors may be used in XML documents, provided all attribute values are quoted, the `A` element and its attributes are declared in the DTD, and all other well-formedness criteria are met. In general, however, XML has better means than named anchors to identify locations.

Relative Location Terms

`id`, `root`, and `html` are absolute location terms. Absolute location terms can find a particular element in a document regardless of what else is in the document. However, more commonly you want to find the first element of a given type, the

last element of a given type, the first child of a particular type, the next element of a given type, all elements of a given type, or something similar. These tasks are accomplished by attaching a relative location term to an absolute location term to form a *compound locator*.

The most general XPointer is a single absolute location term followed by any number of relative location terms. Each term in the list is relative to the one that precedes it, except for the first absolute location term. Terms in the list are separated by periods.

For example, look at the family tree document in Listing 17-1. This fragment selects the first NAME element of the sixth PERSON element in the root element:

```
http://www.theharolds.com/genealogy.xml#root().child(6,PERSON).child(1,NAME)
```

In this example, that's `<NAME>Maria Bellau</NAME>`.

For another example, suppose you want to link to the NAME element of Domeniquette Celeste Baudean. The easiest way to do this is to identify her PERSON element by its ID, p1, then use the child() relative location term to refer to the first (and only) NAME child element, like this:

```
http://www.theharolds.com/genealogy.xml#id(p1).child(1,NAME)
```

This URI says to look at the document `http://www.theharolds.com/genealogy.xml`, find its root element, then find the element with the ID p1, then select its first NAME child.

Although `genealogy.xml` includes ID attributes for most elements, and although they are convenient, they are not required for linking into the document. You can select any element in the document simply by counting down from the root element. Because Maria Bellau's the first person in the document, you can count one PERSON down from the root, then count one NAME down from that. This URI accomplishes that:

```
http://www.theharolds.com/genealogy.xml#root().child(1,PERSON).child(1,NAME)
```

This URI says to look at the document `http://www.theharolds.com/genealogy.xml`, find its root element, then find the first PERSON element that's an immediate child of the root element, and then find its first NAME element.

If no absolute location term is included in the XPointer, then `root()` is assumed. For instance, the previous example could have been written more compactly, like this:

```
http://www.theharolds.com/genealogy.xml#child(1,PERSON).child(1,NAME)
```

You can compress this still further by omitting the second `child` location term (though not its arguments). For example:

```
http://www.theharolds.com/genealogy.xml#child(1,PERSON).(1,NAME)
```

When the term is omitted this way, it is assumed to be the same as the previous term. Because there's no term in front of `.(1, NAME)`, it's assumed to be the same as the previous one, `child`.

There are other powerful selection techniques, which are discussed below. In fact, including `child()`, there are seven relative location terms. These are listed in Table 17-1. Each serves to select a particular subset of the elements in the document. For instance, the `following` relative location term selects from elements that come after the source element. The `preceding` relative location term selects from elements that come before the source element.

Table 17-1
Relative Location Terms

<i>Term</i>	<i>Meaning</i>
<code>child</code>	Selects from the immediate children of the source element
<code>descendant</code>	Selects from any of the content or child elements of the source element
<code>ancestor</code>	Selects from elements that contain the source element
<code>preceding</code>	Selects from elements that precede the source element
<code>following</code>	Selects from elements that follow the source element
<code>psibling</code>	Selects from sibling elements that precede the source element
<code>fsibling</code>	Selects from sibling elements that follow the source element

Because the relative location term alone is generally not enough to uniquely specify which element is being pointed to, additional arguments are passed that further specify the targeted element by instance number, node type, and attribute. The possible arguments are the same for all seven relative location keywords. They are explored in more detail in the “Relative Location Term Argument” section below.

child

The `child` relative location term selects from only the *immediate* children of the source element. For example, consider this URI:

```
http://www.theharolds.com/genealogy.xml#root().child(6,NAME)
```

This points nowhere because there are no `NAME` elements in the document that are direct, immediate children of the root. There are a dozen `NAME` elements that are indirect children. If you'd like to refer to these, you should use the descendant relative locator element instead of `child`.

descendant

The `descendant` relative location term searches through all the descendants of the source, not just the immediate children. For example, `root().descendant(3,BORN)` selects the third `BORN` element encountered in a depth-first search of the document tree. (Depth first is the order you get if you simply read through the XML document from top to bottom.) In Listing 17-1, that selects Louise Pauline Bellau's birthday, `<BORN>29 Oct 1868</BORN>`.

ancestor

The `ancestor` relative location term searches through all the ancestors of the source, starting with the nearest, until it finds the requested element. For example, `root().descendant(2,BORN).ancestor(1)` selects the `PERSON` element, which contains the second `BORN` element. In this example, it selects Elodie Bellau's `PERSON` element.

preceding

The `preceding` relative location term searches through all elements that occur before the source element. The `preceding` locator element has no respect for hierarchy. The first time it encounters an element's start tag, end tag, or empty tag, it counts that element. For example, consider this rule:

```
root().descendant(3,BORN).preceding(5)
```

This says go to Louise Pauline Bellau's birthday, `<BORN>29 Oct 1868</BORN>`, and then move back five elements. This lands on Maria Bellau's `PERSON` element.

following

The `following` relative location term searches through all elements that occur after the source element in the document. Like `preceding`, `following` has no respect for hierarchy. The first time it encounters an element's start tag, end tag, or empty tag, it counts that element. For example, consider this rule:

```
root().descendant(2,BORN).following(5)
```

This says go to Elodie Bellau's birthday, `<BORN>11 Feb 1858</BORN>`, and then move forward five elements. This lands on John P. Muller's `NAME` element, `<NAME>John P. Muller</NAME>`, after passing through Elodie Bellau's `DIED` element, Elodie Bellau's `SPOUSE` element, Elodie Bellau's `PERSON` element, and John P. Muller's `PERSON` element, in this order.

psibling

The `psibling` relative location term selects the element that precedes the source element in the same parent element. For example, `root().descendant(2, BORN).psibling(1)` selects Elodie Bellau's `NAME` element, `<NAME>Elodie Bellau</NAME>`. `root().descendant(2, BORN).psibling(2)` doesn't point to anything because there's only one sibling of Elodie Bellau's `NAME` element before it.

fsibling

The `fsibling` relative location term selects the element that follows the source element in the same parent element. For example, `root().descendant(2, born).fsibling(1)` selects Elodie Bellau's `DIED` element, `<DIED>12 Apr 1898</DIED>`. `root().descendant(2, born).fsibling(3)` doesn't point to anything because there are only two sibling elements following Elodie Bellau's `NAME` element.

Relative Location Term Arguments

Each relative location term begins at a particular place in the document called the *location source*. Generally the location source is indicated by an absolute location term (or the root if no absolute term is specified). You then search forward or backward in the document for the first match that meets specified criteria.

Criteria are given as a list of arguments to the relative location term. These may include the number of elements to search forward or backward, the type of thing to search (element, comment, processing instruction, and so on), and/or the value of an attribute to search. These are given in this order:

1. number
2. type
3. attribute

The number is a positive or negative integer that counts forward or backward from the location source. The type is the kind of thing to count, and the attribute is a list of attribute names and values to match. A relative location term can have a number; a number and a type; or a number, a type, and an attribute list.

The arguments that are present are separated by commas and *no whitespace*. For example:

```
child(1,PERSON,FATHER,p2)
```

The no-whitespace requirement is unusual. It exists so that XPointers can easily be attached to the ends of URLs. For example:

```
http://www.theharolds.com/genealogy.xml#child(1,PERSON,FATHER,p2)
```

If whitespace were allowed, the URLs would have to be x-form-www-url-encoded, like this:

```
http://www.theharolds.com/genealogy.xml#child(1,%20PERSON,%20FATHER,%20p2)
```

For the most part, the same syntax applies to all seven relative location terms.

Selection by Number

The simplest form of selection is by number. The first argument to a relative location term is the index of the node you're pointing at. Positive numbers count forward in the document. Negative numbers count backward. You also can use the `all` keyword to point to all nodes that match the condition.

Number Forward

For instance, in Listing 17-1 the `FAMILYTREE` element is the root. It has 14 immediate children, 12 `PERSON` elements, and two `FAMILY` elements. In order, they are:

```
http://www.theharolds.com/genealogy.xml#root().child(1)
http://www.theharolds.com/genealogy.xml#root().child(2)
http://www.theharolds.com/genealogy.xml#root().child(3)
http://www.theharolds.com/genealogy.xml#root().child(4)
http://www.theharolds.com/genealogy.xml#root().child(5)
http://www.theharolds.com/genealogy.xml#root().child(6)
http://www.theharolds.com/genealogy.xml#root().child(7)
http://www.theharolds.com/genealogy.xml#root().child(8)
http://www.theharolds.com/genealogy.xml#root().child(9)
http://www.theharolds.com/genealogy.xml#root().child(10)
http://www.theharolds.com/genealogy.xml#root().child(11)
http://www.theharolds.com/genealogy.xml#root().child(12)
http://www.theharolds.com/genealogy.xml#root().child(13)
http://www.theharolds.com/genealogy.xml#root().child(14)
```

Greater numbers, such as `http://www.theharolds.com/genealogy.xml#root().child(15)`, don't point anywhere. They're just dangling URLs.

To count all elements in the document, not just the immediate children of the root, you can use `descendant` instead of `child`. Table 17-2 shows the first four descendant XPointers for Listing 17-1, and what they point to. Note especially that `root().descendant(1)` points to the entire first `PERSON` element, including its children, and not just the `PERSON` start tag.

Table 17-2
The First Four Descendants of the Root

<i>XPointer</i>	<i>Points To</i>
<code>root().descendant(1)</code>	<pre><PERSON ID="p1"> <NAME>Domeniquette Celeste Baudean</NAME> <BORN>11 Feb 1858</BORN> <DIED>12 Apr 1898</DIED> <SPOUSE IDREF="p2"/> </PERSON></pre>
<code>root().descendant(2)</code>	<code><NAME>Domeniquette Celeste Baudean</NAME></code>
<code>root().descendant(3)</code>	<code><BORN>11 Feb 1858</BORN></code>
<code>root().descendant(4)</code>	<code><DIED>12 Apr 1898</DIED></code>

Number Backward

Negative numbers enable you to move backward from the current element to the item you're pointing at. In the case of `child` and `descendant`, they count backward from the end tag of the element rather than forward from the start tag. For example, this XPointer selects the element that immediately precedes the element with the ID `f1`:

```
http://www.theharolds.com/genealogy.xml#id(f1).following(-1)
```

In this example, that's the `PERSON` element for **Honore Bellau**. In general, however, your links will be clearer if you avoid negative numbers when possible and use an alternate selector. For example, this selects the same element:

```
http://www.theharolds.com/genealogy.xml#id(f1).preceding(1)
```

In tree-oriented selectors such as `child` and `descendant`, negative numbers indicate that you should count from the end of the parent rather than the beginning. For example, this points at the last `PERSON` element in the document:

```
http://www.theharolds.com/genealogy.xml#root().child(-1,person)
```

This points at the penultimate PERSON element in the document:

```
http://www.theharolds.com/genealogy.xml#root().child(-2,person)
```

Table 17-3 shows the last four descendant XPointers for Listing 17-1, and what they point to. Note that the order in which the elements are entered is now established by the end tags rather than the start tags.

Table 17-3
The Last Four Descendants of the Root

<i>XPointer</i>	<i>Points To</i>
root().descendant(1)	<FAMILY ID="f2"> <HUSBAND IDREF="p7"/> <WIFE IDREF="p6"/> </FAMILY>
root().descendant(2)	<WIFE IDREF="p6"/>
root().descendant(3)	<HUSBAND IDREF="p7"/>
root().descendant(4)	<FAMILY ID="f1"> <HUSBAND IDREF="p2"/> <WIFE IDREF="p1"/> <CHILD IDREF="p3"/> <CHILD IDREF="p5"/> <CHILD IDREF="p6"/> <CHILD IDREF="p8"/> <CHILD IDREF="p10"/> <CHILD IDREF="p12"/> </FAMILY>

all

As well as specifying a number to select, you can use the keyword `all`. This points to all nodes that match a condition. For example, this rule refers to all children of the element with ID `f1`:

```
http://www.theharolds.com/genealogy.xml#id(f1).child(all)
```

In other words, this points to:

```
<HUSBAND IDREF="p2"/>
<WIFE IDREF="p1"/>
<CHILD IDREF="p3"/>
<CHILD IDREF="p5"/>
<CHILD IDREF="p6"/>
<CHILD IDREF="p8"/>
<CHILD IDREF="p10"/>
<CHILD IDREF="p12"/>
```

Selection by Node Type

The above rules chose particular elements in the document. However, sometimes you want to select the fifth `WIFE` or the third `PERSON` while ignoring elements of other types. Selecting these by instance number alone is prone to error if the document changes. The addition or deletion of a single element in the wrong place can misalign all links that rely only on instance numbers.

Occasionally you may want to select processing instructions, comments, CDATA sections, or particular raw text in a document. You can accomplish this by adding a second argument to the relative location term — after the number — that specifies which nodes you're counting and (implicitly) which you're ignoring. This can be the name of the element you want to point to or one of six keywords listed in Table 17-4.

Table 17-4
Possible Second Arguments for Relative Location Terms

<i>Type</i>	<i>Match</i>
<code>#element</code>	Any element
<code>#pi</code>	Any processing instruction
<code>#comment</code>	Any comment
<code>#text</code>	Any nonmarkup character data
<code>#cdata</code>	CDATA sections
<code>#all</code>	All of the above
<i>Name</i>	Elements with the specified name

Most selection rules include the type of the element sought. You've already seen examples where `root().child(6, PERSON)` selects the sixth `PERSON` child of `root`. This may refer to the wrong individual if a `PERSON` element is added or deleted, but at least it is a `PERSON` element instead of something else like a `FAMILY`.

You can also specify just a type and omit the instance number (though not the comma). For example, this URI selects all `PERSON` elements in the document regardless of position:

```
http://www.theharolds.com/genealogy.xml#root().child(,PERSON)
```

Pay special attention to the orphaned comma in front of `PERSON`. It is required by the BNF grammar in the current version of the XPointer specification. Its presence makes it slightly easier for programs to parse the XPointer, even if it makes it harder for humans to read the XPointer.

Exactly what the application does when all `PERSON` elements are targeted is up to the application. In general, something more complex than merely loading the document and positioning it at the targeted element is suggested, since there is more than one targeted element. If the application uses this fragment to decide which parts of a document to load, then it loads all the elements of the specified type.

However, this is unusual. Most of the time, selection by type is only used to further restrict the elements selected until only a single one remains targeted.

Name

The most common use for the second argument to a relative location term is to provide a name for the element type. For instance, suppose you want to point to the first `FAMILY` element that's a child of the root element, but you don't know how it's intermixed with `PERSON` elements. This rule accomplishes that:

```
http://www.theharolds.com/genealogy.xml#root().child(1,FAMILY)
```

This is particularly powerful when you chain selection rules. For example, this points to the second `CHILD` element of the first `FAMILY` element:

```
http://www.theharolds.com/genealogy.xml#root().child(1,FAMILY).  
child(2,CHILD)
```

In fact, it's more common to specify the type of the element you're selecting than not to specify it. This is especially true for relative location terms that don't respect hierarchy such as `following` and `preceding`.

#element

If no second argument is specified, then elements are matched, but processing instructions, comments, `CDATA` sections, character data, and so forth are not matched. You can replicate this behavior with the keyword `#element` as the second argument. For example, these two URIs are the same:

```
http://www.theharolds.com/genealogy.xml#id(f2).preceding(1)  
http://www.theharolds.com/genealogy.xml#id(f2).preceding  
(1,#element)
```

The main reason to use `#element` is so you can then use a third argument to match against attributes.

#text

The `#text` argument selects raw text inside an element. It's most commonly used with mixed content. For example, consider this `CITATION` element from Listing 12-3 in Chapter 12:

```
<CITATION CLASS="TURING" ID="C2">
  <AUTHOR>Turing, Alan M.</AUTHOR>
  "<TITLE>On Computable Numbers,
  With an Application to the Entscheidungs-problem</TITLE>"
  <JOURNAL>
  Proceedings of the London Mathematical Society</JOURNAL>,
  <SERIES>Series 2</SERIES>,
  <VOLUME>42</VOLUME>
  (<YEAR>1936</YEAR>):
  <PAGES>230-65</PAGES>.
</CITATION>
```

The following XPointer refers to the quotation mark before the `TITLE` element.

```
id(C2).child(2,#text)
```

The first text node in this fragment is the whitespace between `<CITATION CLASS="TURING" ID="C2">` and `<AUTHOR>`. Technically, this XPointer refers to all text between `</AUTHOR>` and `<TITLE>`, including the whitespace and not just the quotation mark.



Caution

XPointers that point to text nodes are tricky. I recommend you avoid them if possible, just as you should avoid mixed content. Of course, you may not always be able to, especially if you need to point to parts of documents written by other authors who don't follow this best practice.

Because character data does not contain child elements, further relative location terms may not be attached to an XPointer that follows one that selects a text node. Since character data does not have attributes, attribute arguments may not be used after `#text`.

#cdata

The `#cdata` argument specifies that a `CDATA` section (more properly, the text of a `CDATA` section) is to be selected. For example, this XPointer refers to the second `CDATA` section in a document:

```
root().following(2,#cdata)
```

Because CDATA sections cannot have children, further relative location terms may not be attached to an XPointer that follows one that selects a CDATA section. Since CDATA sections do not have attributes, attribute arguments may not be used after `#cdata`.

#pi

On rare occasions you may want to select a processing instruction rather than an element. In this case, you can use `#pi` as the second argument to the location term. For example, this XPointer selects the second processing instruction in the document's third BEAN element:

```
root().descendant(3,BEAN).child(2,#pi)
```

Because processing instructions do not contain attributes or elements, you cannot add an additional relative location term after the first term that selects a processing instruction. However, you can use a `string()` location term to select part of the text of the processing instruction.

#comment

XPointers point to comments in much the same way they point to processing instructions. The literal `#comment` is used as the second argument to the location term. For example, this XPointer points to the third comment in Listing 17-1:

```
http://www.theharolds.com/genealogy.xml#descendant(3,#comment)
```

Because comments do not contain attributes or elements, you cannot add an additional relative location term after the first term that selects a processing instruction. You can use a `string()` location term to select part of the text of the processing instruction.

#all

On very rare occasions, you may wish to select a particular node in a document regardless of whether it's an element, raw character data, a processing instruction, a CDATA section, or a comment. The only reason I can think of to do this is if you're iterating through all nodes in the document or element. By using `#all` as the second argument to a relative location term, you can ignore the type of the thing you're matching. For example, consider this fragment from Listing 12-3 in Chapter 12:

```
<CITATION CLASS="TURING" ID="C3">
  <AUTHOR>Turing, Alan M.</AUTHOR>
  "<TITLE>Computing Machinery & Intelligence</TITLE>"
  <JOURNAL>Mind</JOURNAL>
  <VOLUME>59</VOLUME>
  (<MONTH>October</MONTH>
  <YEAR>1950</YEAR>):
  <PAGES>433-60</PAGES>
</CITATION>
```

Table 17-5 lists four XPointers that simply count nodes down from the CITATION element. It also lists what is pointed to by the XPointers.

<i>XPointer</i>	<i>Points To</i>
<code>id(C3).following(1,#all)</code>	the whitespace between <code><CITATION CLASS="TURING" ID="C3"></code> and <code><AUTHOR></code>
<code>id(C3).following(2,#all)</code>	<code><AUTHOR>Turing, Alan M.</AUTHOR></code>
<code>id(C3).following(3,#all)</code>	Turing, Alan M.
<code>id(C3).following(4,#all)</code>	"

Selection by Attribute

You can add third and fourth arguments to relative location terms to point to elements by attributes. The third argument is the attribute name. The fourth argument is the attribute value. For example, to find the first PERSON element in the document `http://www.theharolds.com/genealogy.xml` whose FATHER attribute is Jean Francois Bellau (ID p2), you could write:

```
root().child(1,PERSON,FATHER,p2)
```

If you include a third argument, you must include a fourth argument. You can't match against an attribute name without also matching against an attribute value. However, you can use an asterisk for either the name or the value to indicate that anything matches. Setting the third argument to an asterisk (*) indicates that any attribute name is allowed. For example, this XPointer selects all elements that have an attribute value of p2 for any attribute:

```
root().child(all,#element,*,p2)
```

This rule selects the first PERSON element in the document that has an attribute value of p2, regardless of whether that attribute appears as a FATHER, a MOTHER, an ID, or something else.

```
root().child(1,PERSON,*,p2)
```

In Listing 17-1, this is Jean Francois Bellau's PERSON element.

Setting the fourth argument to an asterisk (*) indicates that any value is allowed, including a default value read from the ATTLIST declaration in the DTD. For

example, this rule selects the first element in the document that has a `FATHER` attribute:

```
root().child(1,#element,FATHER,*)
```

In Listing 17-1, this is Elodie Bellau's `PERSON` element.

You can use `#IMPLIED` as the fourth argument to match against attributes that don't have a value, either directly specified or defaulted. For instance, this rule finds the first `PERSON` element that doesn't have a `FATHER` attribute:

```
root().child(1,PERSON,FATHER,#IMPLIED)
```

In Listing 17-1, this is Domeniquette Celeste Baudean's `PERSON` element.

Attribute arguments only work on relative location terms that select an element. You cannot use them when the second argument is `#text`, `#cdata`, `#pi`, or `#comment` because these nodes do not have attributes.

String Location Terms

Selecting a particular element is almost always good enough for pointing into well-formed XML documents. However, on occasion you need to point into non-XML data or XML data in which large chunks of non-XML text is embedded via `CDATA` sections, comments, processing instructions, or some other means. In these cases you may need to refer to particular ranges of text in the document that don't map onto any particular markup element. You can use a string location term to do this.

A string location term points to an occurrence of a specified string. Unlike most other location terms, a string location term can point to locations inside comments, `CDATA`, and the like. For example, this fragment finds the first occurrence of the string "Harold" in Listing 17-1:

```
http://www.theharolds.com/genealogy.xml#string(1,"Harold")
```

This targets the position immediately preceding the *H* in Harold in Charles Walter Harold's `NAME` element. This is not the same as pointing at the entire `NAME` element as an element-based selector would do.

You can add an optional third position argument to specify how many characters to target to the right of the beginning of the matched string. For example, this targets whatever immediately follows the first occurrence of the string "Harold" because *Harold* has six letters:

```
http://www.theharolds.com/genealogy.xml#string(1,"Harold",6)
```

An optional fourth argument specifies the number of characters to select. For example, this URI selects the first occurrence of the entire string “Harold” in Listing 17-1:

```
http://www.theharolds.com/genealogy.xml#string(1,"Harold",1,6)
```

Use the empty string (“”) in a string location term to specify particular characters in the document. For example, the following URI targets the 256th character in the document. (To be precise, it targets the position between the 255th and 256th element in the document.)

```
http://www.theharolds.com/genealogy.xml#string(256, "")
```

When matching strings, case and whitespace are considered. Markup characters are ignored.

Instead of requesting a particular instance of a particular string match, you can ask for all of them by using the keyword `all` as the first argument. For example, this rule selects all occurrences of the string “Bellau” in the document:

```
http://www.theharolds.com/genealogy.xml#string(all,"Bellau")
```

This can result in a noncontiguous selection, which many applications may not understand, so use this technique with caution.

The origin Absolute Location Term

The fourth absolute location term is `origin`. However, it’s only useful when used in conjunction with one or more relative location terms. In intradocument links, that is, links from one point in a document to another point in the same document, it’s often necessary to refer to “the next element after this one,” or “the parent element of this element.” The `origin` absolute location term refers to the current element so that such references are possible.

Consider Listing 17-2, a simple slide show. In this example, `origin().following(1,SLIDE)` refers to the next slide in the show. `origin().preceding(1,SLIDE)` refers to the previous slide in the show. Presumably this would be used in conjunction with a style sheet that showed one slide at a time.

Listing 17-2: A slide show

```
<?xml version="1.0"?>
<SLIDESHOW>
  <SLIDE>
    <H1>Welcome to the slide show!</H1>
    <BUTTON xml:link="simple">
```

```

        href="origin().following(1,SLIDE)">
    Next
  </BUTTON>
</SLIDE>
<SLIDE>
  <H1>This is the second slide</H1>
  <BUTTON xml:link="simple"
    href="origin().preceding(1,SLIDE)">
    Previous
  </BUTTON>
  <BUTTON xml:link="simple"
    href="origin().following(1,SLIDE)">
    Next
  </BUTTON>
</SLIDE>
<SLIDE>
  <H1>This is the second slide</H1>
  <BUTTON xml:link="simple"
    href="origin().preceding(1,SLIDE)">
    Previous
  </BUTTON>
  <BUTTON xml:link="simple"
    href="origin().following(1,SLIDE)">
    Next
  </BUTTON>
</SLIDE>
<SLIDE>
  <H1>This is the third slide</H1>
  <BUTTON xml:link="simple"
    href="origin().preceding(1,SLIDE)">
    Previous
  </BUTTON>
  <BUTTON xml:link="simple"
    href="origin().following(1,SLIDE)">
    Next
  </BUTTON>
</SLIDE>
...
<SLIDE>
  <H1>This is the last slide</H1>
  <BUTTON xml:link="simple"
    href="origin().preceding(1,SLIDE)">
    Previous
  </BUTTON>
</SLIDE>
</SLIDESHOW>

```

Generally, the `origin()` location term is only used in fully relative URIs in XLinks. If any URI part is included, it must be the same as the URI of the current document.

Spanning a Range of Text

In some applications it may be important to specify a range of text rather than a particular point in a document. This can be accomplished via a *span*. A span begins at one XPointer and continues until another XPointer.

A span is indicated by the keyword `span()` used as a location term. However, the arguments to `span()` are two location terms separated by a comma identifying the beginning and end of the span. If these are relative location terms, then the term preceding the span is the source for both terms.

For example, suppose you want to select everything between the first `PERSON` element and the last `PERSON` element in `genealogy.xml`. This XPointer accomplishes that:

```
root().span(child(1,PERSON),child(-1,PERSON))
```

Summary

In this chapter you learned about XPointers. In particular you learned:

- ♦ XPointers refer to particular parts of or locations in XML documents.
- ♦ The `id` absolute location term points to an element with a specified value for an ID type attribute.
- ♦ The `root` absolute location term points to the root element of an XML document.
- ♦ The `html` absolute location term points to a named anchor in an HTML document.
- ♦ Relative location terms can be chained to make more sophisticated compound selectors. The term to which a term is relative is called the location source.
- ♦ The `child` relative location term points to an immediate child of the location source.
- ♦ The `descendant` relative location term points to any element contained in the location source.
- ♦ The `ancestor` relative location term points to an element that contains the location source.
- ♦ The `preceding` relative location term points to any element that comes before the location source.
- ♦ The `following` relative location term points to any element following the location source.

- ♦ The `psibling` relative location term selects from sibling elements that precede the target element.
- ♦ The `fsibling` relative location term selects from sibling elements that follow the target element.
- ♦ Each relative location term has between one and four arguments: a number, a type, an attribute name, and an attribute value.
- ♦ The first argument to a relative location term is a number determining the relative position of the targeted node or the keyword `all`.
- ♦ The second argument to a relative location term determines the type of the targeted node and may be the name of the element or one of the keywords `#element`, `#pi`, `#comment`, `#text`, `#cdata`, `#all`.
- ♦ The third argument to a relative location term determines the name of the attribute possessed by the targeted node.
- ♦ The fourth argument to a relative location term determines the value of an attribute of the targeted node.
- ♦ The `string` location term points to a specified block of text in the location source.
- ♦ The `origin` absolute location term points to the current element.
- ♦ Spans refer to a range of text instead of merely one particular element.

The next chapter explores namespaces. Namespaces use URIs as a means of sorting out the elements in a document that's formed from multiple XML applications. For example, namespaces allow you to simultaneously use two different XML vocabularies that define the same elements in incompatible ways.



